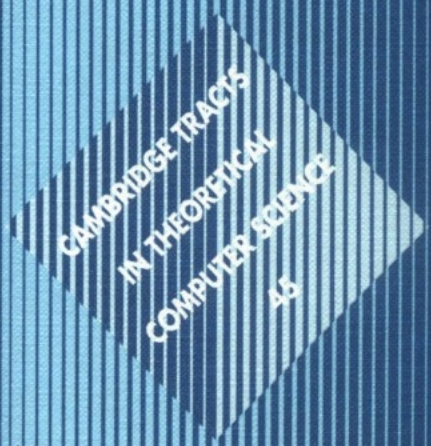
The Optimal Implementation of Functional Programming Languages

Andrea Asperti
and Stefano Guerrini



PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge CB2 1RP, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK http://www.cup.cam.ac.uk
40 West 20th Street, New York, NY 10011-4211, USA http://www.cup.org
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First published 1998

Printed in the United Kingdom at the University Press, Cambridge

Typeset in Computer Modern 10/13pt, in LaTeX 2ε

A catalogue record of this book is available from the British Library

Library of Congress cataloguing in Publication data

Asperti, Andrea.
The optimal implementation of functional programming languages /
Andrea Asperti, Stefano Guerrini.
p. cm.
Includes bibliographical references and index.
ISBN 0 521 62112 7 (hb)
1. Functional programming languages. I. Guerrini, Stefano, 1965–.
II. Title.
QA76.62.A84 1998
005.13–dc21 97-51550 CIP

ISBN 0 521 62112 7 hardback

The Optimal Implementation of Functional Programming Languages

Andrea Asperti

Stefano Guerrini

Contents

1
Introduction

This book is about *optimal sharing* in functional programming languages.

The formal notion of sharing we shall deal with was formalized in the seventies by Lévy in terms of "families" of redexes with a same origin—more technically, in terms of sets of redexes that are residuals of a unique (virtual) redex. The goal of his foundational research was to characterize formally what an *optimally efficient* reduction strategy for the λ-calculus would look like (even if the technology for its implementation was at the time lacking). Lévy's dual goals were *correctness*, so that such a reduction strategy does not diverge when another could produce a normal form, and *optimality*, so that redexes are not duplicated by a reduction, causing a redundancy in later calculation [Lév78, Lév80]. The relevant functional programming analogies are that call-by-name evaluation is a correct but not optimal strategy, while call-by-value evaluation is a (very rough) approximation of an incorrect but "optimal" strategy.

Lévy clearly evinced that no sharing based implementation of λ-calculus could reduce in a single step two redexes belonging to distinct families. So, according to his definition, a reduction technique is *optimal* when (*i*) a whole family of redexes is contracted in a single computational step and (*ii*) no unneeded work is ever done (i.e., the reduction does never contract a redex that would be erased by a different reduction strategy).

Such optimal and correct implementations were known for recursion schemas [Vui74], but not for ones where higher-order functions could be passed as first-class values. Actually, all the naive algorithms suggested by Lévy's theory looked quite artificial and required an unfeasible overhead to decide whether two redexes belong to a same family.

Recent research by Lamping has shown that there indeed exist λ-calculus evaluators satisfying Lévy's specification [Lam89, Lam90]. The

fact that more than ten years elapsed between Lévy's definition of optimality and Lamping's reduction algorithm should already give a gist of the complexity of the problem. As a matter of fact, having a unique representation for all the redexes in a same family entails the use of a *very sophisticated* sharing technique, where all the computational and expressive power of higher-order languages really shines. Note in particular that two redexes of a same family can nest one into the other (for instance, given $R = (\lambda x.M\,N)$, its subterms M and N may contain redexes in the family of R). Therefore, all the traditional implementation techniques for functional languages (mostly based on supercombinators, environments or continuations) fail in avoiding useless repetitions of work: no machinery in which the sharing is exploited at the (first order) level of subterms can be Lévy's optimal.

Lamping's breakthrough was a technique to share contexts, that is, to share terms with an unspecified part, say a hole. Each instance of a context may fill its holes in a distinct way. Hence, for any shared context, there are several access pointers to it and, for each hole, a set of possible choices for how to fill it—a choice for each instance of the context. Lamping's solution moved from the idea that some control operator should manage the matching between the instances of a context and the ways in which its holes may be filled. As a result, Lamping extended λ-graphs by two sharing nodes called fans: a "fan-in" node collecting the pointers to a context; a "fan-out" node collecting the ways in which a hole can be filled (the exit pointers from the context). Assuming that each pointer collected by a fan-in or fan-out is associated to a named port, the correspondence between entering and exiting pointers is kept provided to have a way to pair fan-in's and fan-out's: the instance of the context corresponding to the pointer connected to a fan-in port with name a fills its holes with the subgraphs accessed through the ports with name a of the matching fan-out's.

Let us try to explain in more detail the last point by a simplified example. Let us take the language of named angles $L ::= \rangle_x \mid {}^x\langle$, where x is a symbol chosen over a given alphabet. In a string $a\rangle_x b\,{}^x\langle c$, the pair of angles with the same label x is matching when the corresponding pair of parentheses matches (by the usual rules) in the expression $a'(b')c'$ obtained by erasing the angles labeled by $y \neq x$ and replacing "(" for $\rangle_x$ and ")" for ${}^x\langle$. According to this definition of matching, two substrings enclosed between matching angles may even overlap, as in $\rangle_x\rangle_y\,{}^x\langle{}^y\langle$. The string between two matching angles is the idealization of a shared context: a left angle $\rangle_x$ denotes the access point to a shared context, a right

angle $_x\langle$ denotes an exit point from the context (i.e., a hole). Assuming that fans are binary operators, at the left angle we enter the shared part from one of two possible choices (top or down for instance); at a right angle we have instead two exit ports, one for each way in which the context can be accessed. The key idea is that the choice between the two possibilities at a right angle (fan-out) is forced by the matching: entering by the top/down port of a fan-in we have to exit by the top/down port of any matching fan-out. In the implementation of λ-calculus, a naive attempt to keep track of the border of the shared parts based on the previous labeling technique (for more details see the example developed in section 2.2) corresponds in L to have a rewriting system whose rules may copy and/or move angles but not change their names. Unfortunately, such a solution is not adequate, as there are cases in which two distinct shared parts have the same origin and then the same label (a detailed example will be given at the beginning of Chapter 3). For instance, we could get cases as $\rangle_x\rangle_x{}^{x}\langle^{x}\langle$ in which the correct labeling corresponds to $\rangle_{x'}\rangle_{x''}{}^{x'}\langle^{x''}\langle$.

The actual achievement of Lamping is the use of a dynamic labeling of fans controlled by other nodes called brackets. Going on with our simplified example, let us assume that the labels are integers and that we enclose into brackets regions in which the labels of the angles have to be incremented by 1. That is, $\rangle_n\,[\,\rangle_n\,]\,{}^{n}\langle\,[\,{}^{n}\langle\,]$ is equivalent to $\rangle_n\rangle_{n+1}{}^{n}\langle^{n+1}\langle$. The use of such brackets allows us to deal with the cyclic structures that may arise along the reduction. For instance, reminding that two shared parts may be even nested, it is immediate to see that in order to share them we need some involved cyclic structure. In our simplified example, assuming a correspondence between our strings and paths of the shared graph, cycles would correspond to strings containing multiple occurrences of the angles associated to a same node. However, exploiting the fact that the two occurrences of such a node have two distinct access paths, we may maintain the correct matching using the brackets to ensure that two angles (fans) match only when the labels computed taking into account the brackets coincide. By the way, this is only a starting point. In fact, see Chapter 3, also the brackets must be labeled by an index and have a problem of matching.

A surprising and remarkable aspect of Lamping's algorithm (and its semantics) is its tight relation to Girard's Linear Logic and in particular with that deep analysis of the mathematical foundations of operational semantics that goes under the name of "Geometry of Interaction" (GOI). In particular, our previous example can be directly reformulated in the

language of GOI. As a matter of fact, GOI is essentially based on an algebraic interpretation of graph paths: a weight is associated to each path forcing to 0 the value of the wrong ones. Namely, in the dynamic algebra of GOI there are two symbols r, s and their duals $\bar{r}, \bar{s}$ corresponding to our angled parentheses—the pair r and s corresponds to the pair top and down. The axioms are: $\bar{r}r = 1 = \bar{s}s$ and $\bar{r}s = 0 = \bar{s}r$. In practice, a wrong bracketing yields 0, while a correct and fully balanced bracketing yields 1; the other cases correspond to correct expressions with some unbalanced parentheses. According to this we have $\bar{r}\bar{s}sr = 1$ and $\bar{r}\bar{s}rs = 0$. The reindexing we denoted by brackets correspond in GOI to an operation "!" for which $x!(P) = !(P)x$ and $!(PQ) = !(P)!(Q)$, where x is either r or s. Hence, $!(\bar{r})\bar{s}!(r)s = 1$, since $!(1) = 1$.

In general, the study of optimal reduction has provided important insights and connections between linear logic, control structures, context semantics, geometry of interaction, intensional semantics. Other relations, in particular with game semantics and the area of full abstraction look at hand, but have not been really exploited, yet. This is actually the most intriguing nature of optimality: a wonderfull, tough mathematical theory applied to a very practical problem: sharing. And this is also the leading theme of the book, which intends to cover optimal implementations issues from the most theoretical to the most pragmatical ones.

1.0.1 How to read this book

The structure of the book is quite complex. Since, probably, *it is not a good idea to read it sequentially* (unless you really wish to become an expert in this field) we feel compelled to add some "reading instructions" (we usually detest them).

First of all let us briefly explain the content of each chapter.

Chapter 2: Optimal Reduction This chapter provides a hopefully friendly approach to the (pragmatical) problem of optinal sharing. Several sharing mechanisms are discussed (Wadsworth's graph reduction, combinators and environment machines), and Lamping's "abstract" reduction algorithm is introduced.

Chapter 3: The Full Algorithm Here we introduce all the technology of sharing graphs, and its context-semantics. You can skip section 3.5 (correctness) at first reading.

Chapter 4: Optimal Reduction and Linear Logic A naive intro-

duction to the relation between Optimality and Linear Logic. You can definetely skip it if you are not interested in this topic. However, in its simplicity, it may offer you an alternative intuition on Optimal Reduction (or, conversely, a simple introduction to Linear Logic, in case you are not acquainted with it).

Chapter 5: Redex Families and Optimality The formal definition of optimality, at last! Why did we postpone it up to Chapter 5? Well, the book is mainly addressed to Computer Scientists, which are supposed to have some clear intuition of the problem of sharing. You do not really need the formal definition of family in order to appreciate the beauty and the expressiveness of sharing graphs. But, and here things start getting really intriguing, this reduction technique relies on an even more beatifull and compelling mathematical theory!

Chapter 6: Paths The way we give semantics to sharing graphs is by reading paths inside them. But this kind of path semantics has been introduced for the first time in a totally different setting: the geometry of interaction of Linear Logic. Any relation? Read this chapter if you want to learn the answer.

Chapter 7: Read-back . In this chapter we will pursue a more syntactical study of the properties of sharing graphs and we will present a way to group control nodes allowing to solve and simplify the problem of read-back.

Chapter 8: Other translations in sharing graphs Sharing graphs are the real thing. But you may encode λ-terms into sharing graphs in a number of different ways. Here we just mention a few of them, relating them by the so called "bus notation".

Chapter 9: Safe Rules The most annoying problem of optimal reduction is the accumulation of spurious book-keeping information. In this chapter we discuss this problem and give a partial, but practical solution.

Chapter 10: Complexity Have a glance at this chapter as soon as you can. It's not the complexity result that matters, at first reading (optimal β-reduction is not elementary recursive), but the examples of sharing graphs, and their reduction.

Chapter 11: Functional Programming So far, we have been in the comfortable setting of λ-terms. Now, we start adding a few "δ-rules".

Chapter 12: The Bologna Optimal Higher-order Machine Yes, it

works! A real, optimal implementation of a core functional language, freely available by ftp.

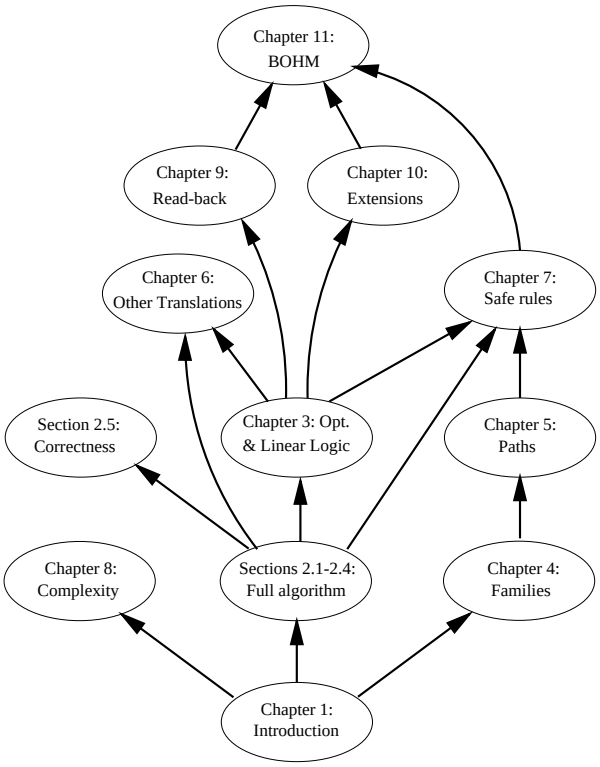


Fig. 1.1. Dependency graph of the book.

Now, there are several different "access paths" to all the material covered by this book, depending on your interests and cultural formation. We just mention a few possibilities.

First reading If you are just interested to understand what all this story of optimality is about, and to have a closer look at sharing graphs, we suggest to start with the introduction to Optimal Reduction in Chapter 2 (this chapter is obviously *compulsory* in every "access paths"), jumping then to Chapter 11, where a lot of interesting examples of sharing graphs are discussed (that should be enough to get a gist of the actual power of this formalism). If you get intrigued by the problem of the correct

matching of fan, you can then have a look at the first sections of Chapter 3, up to section 3.3 or 3.4 (section 3.5 is rather technical and it can be skipped at first reading!). On the other side, if you are interested in the formal definition of optimal sharing, please glance at Chapter 5.

Implementation If you are interested in implementative issues, and you wish to understand what are the open problems in this area, you must eventually read sections 3.1–3.4 (again, you can skip section 3.5). Then, jump to Chapter 8, and read sequentially the remaining part of the book.

λ-calculus So, you are an impenitent theorician, you don't know anything about sharing (maybe you do not even care), but people told you it could be worth to have a look at this book. Good. You had probably better start with Chapter 5 (the formal definition of optimality), and 6 (paths), coming back to Chapter 3 for all the references to sharing graphs and optimal reduction. When you feel ready, you may start tackle section 3.5 (correctness) and Chapter 7 (read-back).

Linear Logic You are interested in linear logic, and you wish to understand what is the relation with optimality. For an "elementary" introduction, start reading Chapter 4 (this requires Chapter 3 as well). However, the real staff is contained in Chapter 6 (paths), and Chapter 7 (read-back). A discussion of the potential relevance of additive types in the setting of optimality is also contained in section 12.4 (problems).

2
Optimal Reduction

In the λ-calculus, whenever we execute a β-reduction $(\lambda x.M\,N) \to M[N/x]$ we replace each occurrence of the bound variable x in M by a copy of the argument N. This operation could obviously duplicate work, since any reduction required to simplify the argument will be repeated on each copy. At first glance, it could seem that choosing a innermost reduction strategy (i.e., reducing arguments first) we would easily solve this problem, simply because we would avoid copying redexes. Unfortunately, this is not true. First of all, we should also avoid to reduce useless redexes (i.e., by the standardization theorem, redexes not reduced in the leftmost-outermost reduction), but they cannot be found effectively in the lambda-calculus. Moreover, and this is the crucial point, even a reduction strategy that always reduces *needed* internal redexes is not guaranteed to reach the normal form (if it exists) in a minimal number of steps. For example, in the case of the λI-calculus, where all redexes are needed for computing the normal form, innermost reductions could not be optimal. Take for instance the term $M = (\lambda x.(x\,I)\,\lambda y.(\Delta\,(y\,z)))$, where $I = \lambda w.w$ and $\Delta = \lambda x.(x\,x)$. The normal form of M is $(z\,z)$. Some reduction strategies for M are shown in Figure 2.1. In particular, the internal strategies (on the right in the figure) are not the shortest ones. The reason is that they duplicate a subexpression $(y\,z)$ which is not yet a redex, but that will become a redex later on, when y will be instantiated to the identity (by the outermost redex). Note that the redex $(I\,z)$ is intuitively sharable, and it is actually shared by the reduction on the left in Figure 2.1.

A deeper analysis of the previous example allows us to introduce the core idea of optimal reductions: even if the subterm $(y\,z)$ of M is not a redex, reducing M the term I will substitute y creating a redex $(I\,z)$. In a certain sense, the subexpression $(y\,z)$ is a "virtual redex". More

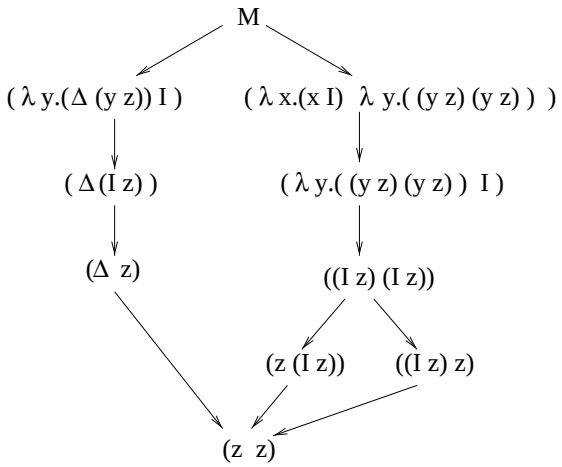


Fig. 2.1. $M = (\lambda x.(x\,I)\,\lambda y.(\Delta\,(y\,z)))$

accurately, we could uniquely describe the virtual redex corresponding to $(y\,z)$ as a path of the abstract syntax tree of M leading from the application in $(y\,z)$ to the λ of the identity, as shown in Figure 2.2.

To find the path of the virtual redex $(y\,z)$ let us proceed as follows. Leaving the @ node of $(y\,z)$ through its left edge a, we would immediately find a redex if at the other end of a there would be a λ node. In this case, there is a variable y. We have not yet a redex but it could become a redex if y get substituted by a λ-abstraction, that is if the binder of y is the left part of a (virtual!) redex having as right part a λ-abstraction. So, let us push in a stack the initial question about y, and proceed asking if λy is the left part of a virtual redex. To this purpose, let us move back from λw to the above application following the edge b, and then let us go down through the edge c. The edge c is a redex and the variable bound by λx occurs as left argument of an @ node. The path bcd is then a virtual redex: it corresponds to the term $(x\,\lambda w.w)$, in which after the contraction of c the abstraction $\lambda y.(\Delta\,(y\,z))$ will take the place of the variable x. Popping from the stack our initial question about y, we see thus that the path $abcde$ gives a connection between the @ node of $(y\,z)$ and the root of a subterm that will replace y after some reductions. Therefore, since the edge e reaches a λ-abstraction, the path $abcde$ corresponds to the virtual redex $(y\,z)$. Moreover, note that, if we imagine that bound variables are explicitly connected to their

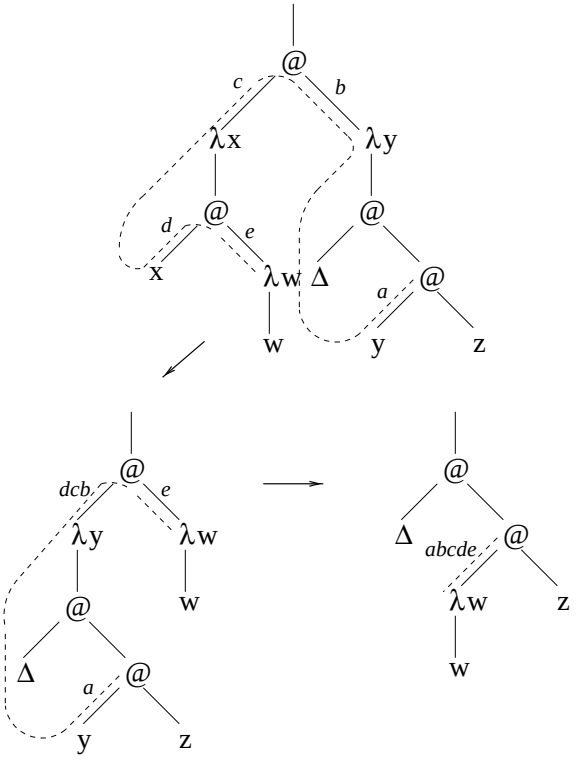


Fig. 2.2. Virtual redexes and paths.

binders (an idea that goes back to Bourbaki), this "path" is an actual path in the graph representation of M.

A complete definition of the "paths" corresponding to virtual redexes will be given in Chapter 6. Here, let us note that, by means of a simple labeling technique, we have been able to label each edge created along the reduction by a corresponding path in the initial term, and that, intuitively, the edge was exactly created by "contraction" of this path. For instance, the edge dcb in Figure 2.2 is the result of the contraction of the corresponding path (note the order of the labels!), while the redex edge abcde is the contraction of the path of the virtual redex (y z).

To implement optimal reductions we must avoid the duplication of explicit and virtual redexes. Pursuing the analogy with paths, we have to find some reduction technique that compute the normal form of a

term without duplicating (or computing twice) *any path corresponding to a virtual redex.*

The constraint to never duplicate a path reflects in an ordering constraint between redexes. For instance, in the previous example it entails to postpone the innermost redex $(\Delta\,(y\,z))$ after the reduction of the (virtual) redex (whose path is) contained in $abcde$. Namely, it means to start with the outermost redex of M, to reduce the redex $(I\,z)$ created by it and, only at this point, to contract the redex involving Δ. Such a reduction strategy corresponds to the one on the left-hand side in Figure 2.1, which in this case is also the optimal one. But what it would have happened if the outer-most redex had duplicated its argument at is turn? The ordering between redexes suggested by the non-duplication constraint of virtual redex paths would have been unsatisfiable.

Lévy gave a clear example of how the effect of duplication of needed redexes can combine in a very subtle and nasty way. Let us define the following λ-terms:

$$\begin{aligned}
\Delta_n &= \lambda x.(\underbrace{x\,x\ldots x}_{n\text{ times}}) \\
F_n &= \lambda x.(x\,I\,\underbrace{x\,x\ldots x}_{n\text{ times}}) \\
G_n &= \lambda y.(\Delta_n\,(y\,z)) \\
\\
G'_n &= \lambda y.(\underbrace{(y\,z)\,(y\,z)\ldots(y\,z)}_{n\text{ times}})
\end{aligned}$$

The reduction of the term $P = (F_m\,G_n)$ is depicted in Figure 2.4. The sequence on the left-hand side, whose length is $n+3$ is a typical inner-most reduction. The one on the right-hand side follows instead an outer-most strategy and requires $m+4$ β-reductions. The shortest reduction strategy depends thus from the value of m and n.

It is remarkable that, in this case, *every reduction strategy duplicates work!* Indeed, the strategy on the right-hand side shares the reduction of G_n to G'_n, but not the reduction of $(I\,z)$ to z. The converse happens for the reduction strategy on the left-hand side.

Exercise 2.0.1 Try to guess the paths of the virtual redexes of $P_{2,2} = (F_2\,G_2)$. Show that any contraction of a redex of $P_{2,2}$ causes the duplication of a virtual redex observing that it causes the duplication of a corresponding path.

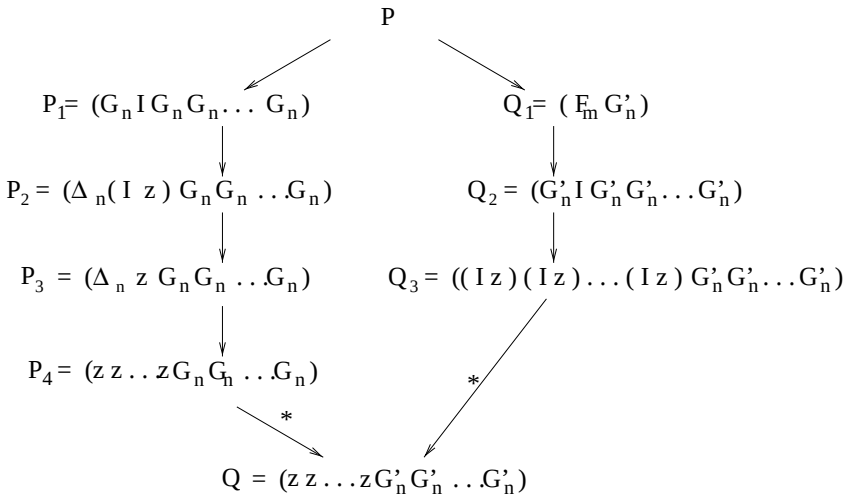


Fig. 2.3. $P = (F_m\ G_n)$

Besides, supposing to have an optimal implementation where no duplication is ever done, what could we say about the reduction of P? In such an optimal machine, all the reductions of P should be equivalent. In fact, such an implementation would correspond to reduce at each step all the copies of a given redex. The term P has only *four* distinguished redexes. Therefore, the reductions of such an optimal machine would correspond to the diagram of Figure 2.4.

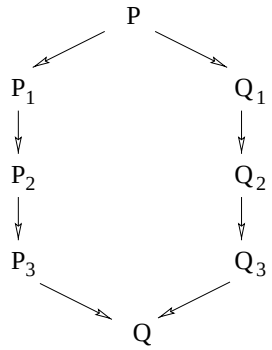


Fig. 2.4. $P = (F_m\ G_n)$

Lévy's example proves not only that the problem of optimal reduc-

tion cannot be simply solved by the choice of a suitable reduction strategy, but also that, if an implementation without duplication exist, its expected performance would be arbitrarily better than any reduction technique without optimal sharing.

Another interesting example is diffusely discussed by Lamping [Lam89]. Let $R = \lambda g.(g(g\,I))$, and take $S = (R\,\lambda h.(R\,(h\,I)))$ (it is easy to see that S reduces to the identity). Again, we are in the same situation. If the outermost redex is reduced first, we eventually create two copies of the innermost one. Conversely, if we start executing the innermost redex, we duplicate the "virtual redex" $(h\,I)$ (that requires the firing of the outermost redex, in order to become an actual redex). In particular, four copies of the application $(h\,I)$ will need to be simplified, if the inner redex is reduced first, compared to only two copies if the outermost redex is reduced first (one for each distinct value of the argument h).

All these examples might look quite artificial, and one could easily wonder what is the actual and practical relevance of optimal implementation techniques. From this respect, it is probably important to stress some other essential properties of optimality.

The only reason because in the λ-calculus different reduction strategies have different lengths is the duplication of redexes (as long as we take care to reduce "needed" redexes only). Supposing to have an implementation technique that avoids these duplications, all reduction strategies would be absolutely equivalent. More precisely, the rewriting system would satisfy a one-step diamond property. Since even in the case of optimality the leftmost-outermost reduction does only reduce needed redexes, it simply jumps over the (undecidable!) problem of establishing the shortest reduction strategy. In particular, an optimal reduction technique is able to combine all the benefits of lazy and strict strategies.

Moreover, and this is a crucial point, an optimal implementation implicitly solves the well known problem of "computing inside a λ" (almost all implementations of functional languages circumvent this problem adopting a weak notion of value that prevents any sharing inside a λ). The relevance of this point should be correctly understood. Consider for instance the λ-term $(\underline{n}\,\underline{2}\,I\,a)$, where $\underline{n}$ and $\underline{2}$ are the Church integers of n and 2, and a is some constant. Note that $(\underline{2}\,I) \rightarrow I$. So, adopting an innermost reduction strategy, we get the normal form a in a number of β-reduction steps linear in n. However, even in typical strict functional languages (like SML or CAML), the computation of $(\underline{n}\,\underline{2}\,I\,a)$ is exponential in n. The reason is that the reduction of $(\underline{2}\,I)$ stops at the weak head normal form, namely (some internal representation of) $\lambda y.(I(Iy))$.

The two applications of the identity are not reduced, and they will be eventually duplicated when this term is passed as an argument to the successive $\underline{2}$. At the end, we are left with 2^n applications of the identity to be reduced. The situation is even worse in lazy languages, such as Scheme, Haskell or Miranda.

The usual objection of the old, week-evaluation school of functional programming is that:

"stopping reduction under a lambda makes good practical sense as it prevents problems associated with reduction of *open* terms.†"

Sure, as well as Pascal makes good practical sense as it prevents problems associated with returning functions as results. But the purpose of Computer Science, if any, is just to study and possibly to solve problems, not just "preventing" them, isn't it?

2.1 Some Sharing mechanisms

By the previous discussion, it is clear that no satisfactory notion of sharing can be obtained without leaving the syntactical universe of λ-expressions and the usual notion of reduction strategy. In other words, we have to look for some representation of λ-terms inside different structures with some (explicit or implicit) sharing mechanism.

2.1.1 Wadsworth's technique

A first attempt in this direction was provided by Wadsworth's graph reduction technique [Wad71]. Wadsworth suggested to represent λ-expressions as directed acyclic graphs similar to abstract syntax trees plus pointers for implementing sharing. In Wadsworth's graphs, identical subtrees (identical subexpressions) are represented by a single piece of graph. As a consequence, the reduction of a β-redex $\lambda x.(M\,N)$ does not require copying the argument N: each occurrence of x in M is simply linked to N. Moreover, any simplification in a shared section of a graph corresponds to multiple simplifications in the corresponding λ-term.

For instance, the Wadsworth graph normal (leftmost-outermost) reduction of the term $M = (\Delta\,\lambda x.((x\,y)\,I))$ is represented in Figure 2.5.

† Anonymous referee.

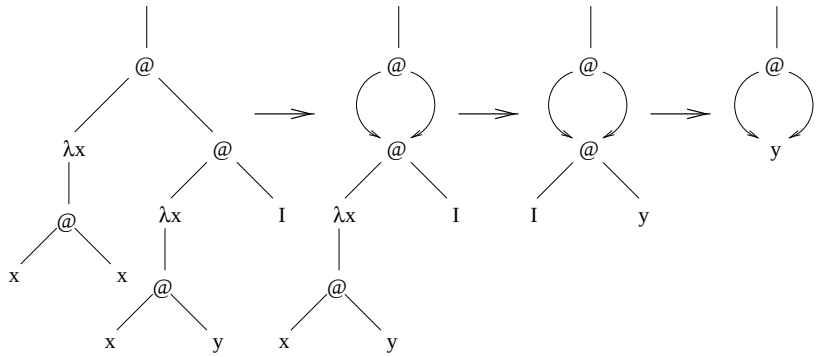


Fig. 2.5. Wadsworth's reduction

The corresponding reduction in the usual syntactical framework of λ-terms is:

$$M \to (\lambda x.((x\,y)\,I)\,\lambda x.((x\,y)\,I)) \xrightarrow{*} ((I\,y)\,(I\,y)) \xrightarrow{*} (y\,y)$$

We see that the second and third step simultaneously contract both the copies of the redexes $\lambda x.((x\,y)\,I)$ and $(I\,y)$ contained in the corresponding terms.

Unfortunately, copying is required also in Wadsworth's algorithm as soon as the λ-term involved in a redex is represented by a shared piece of graph. Otherwise, the substitution called for by the β-reduction would instantiate other instances of the body of the function which have nothing to do with the current redex. Consider for instance the λ-term

$$N = (\lambda x.((x\,y)\,(x\,z))\,\lambda w.(I\,w))$$

After the outermost β-reduction, the argument $\lambda w.(I\,w)$ is shared by the two occurrences of x. At the following step, only the first instance of $\lambda w.(I\,w)$ should be involved in the redex $(\lambda w.(I\,w)y)$. Wadsworth's algorithm proceed thus to the duplication of the functional argument of the redex. But in this way, it duplicates the internal redex $(I\,w)$, that on the contrary should be shared in the two instances. Despite this example could be solved by reducing the functional argument first, in the general case, the problem cannot be fixed just changing the reduction strategy,(as we already remarked at the beginning of the chapter. The reason is that the applications inside the functional argument could be "virtual redexes" (suppose that in the place of I there be a variable bound externally to $\lambda w.(I\,w)$, and note that we are not following a

leftmost-outermost strategy any more). The duplication of such "virtual redexes" (i.e., of the application $(y\,w)$) is as useless and expensive as the duplication of real redexes. As an instructive example, we suggest the reader to try Wadsworth graph reduction technique on Lévy's example in Figure 2.1. He will immediately realize that no reduction strategy is optimal, working with Wadsworth's structures.

2.1.2 Combinators

Graph reduction can be easily adapted to work with *combinators*, instead of λ-expressions. Combinators (either of Combinatory Logic or Super-Combinators) are used to split a β-reduction step into a sequence of simpler (first order) reductions, that essentially correspond to an explicit modeling of the substitution process. While Wadsworth's β-reduction always requires the duplication of the function body, the corresponding combinator reduction is typically able to avoid copying all the subexpressions that do not contain the variable bound by the λ. Nevertheless, this is not enough to avoid duplication of work.

Let us consider a simple example for the Combinatory Logic (CL) case. A typical encoding of λ-abstraction into CL is defined by the following rules:

$$\begin{aligned} \lambda^* x.x &= \mathbf{SKK} \\ \lambda^* x.M &= \mathbf{K}M \qquad \text{if } x \notin \mathtt{var}(M) \\ \lambda^* x.(M\,N) &= \mathbf{S}(\lambda^* x.M)(\lambda^* x.N) \end{aligned}$$

A λ-term is then inductively translated into CL as follows:

$$\begin{aligned} [x]_{CL} &= x \\ [(M\,N)]_{CL} &= [M]_{CL}\,[N]_{CL} \\ [\lambda x.M]_{CL} &= \lambda^* x.[M]_{CL} \end{aligned}$$

Let us take the λ-term $M = (\lambda x.(x(x\,y))\ \lambda z.(A\,B))$. Applying the previous rules, we get $[\lambda z.(A\,B)]_{CL} = \mathbf{S}(\lambda^* z.[A]_{CL})(\lambda^* z.[B]_{CL})$. We see that after the translation, the internal application $(A\,B)$ is split and the two terms $[A]_{CL}$ and $[B]_{CL}$ become the arguments of the combinator **S**. After a few reduction steps, we eventually come to the graph in Figure 2.6.

At this point, the portion of graph enclosed in the dotted line must be duplicated (usually, this duplication step comes together with the reduction of the leftmost-outermost instance of **S**, so it is not always

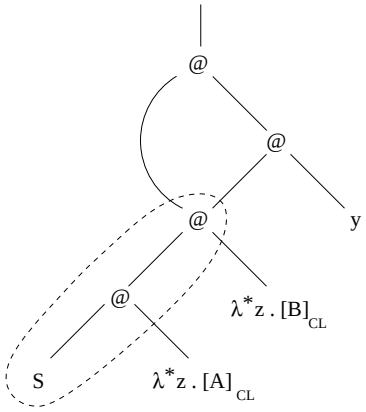


Fig. 2.6. Combinatory Logic reduction

explicit). Unfortunately, this duplication amounts to a duplication of the application $(A\,B)$ in the corresponding λ-term. If this application was a redex (either actual or virtual), we definitely loose the possibility of sharing its reduction.

Let us consider another example. Let $M = (\underline{2}\,I)$ (where $I = \mathbf{SKK}$). In Combinatory Logic, this term does not reduce to I, but to the term $\mathbf{S}(\mathbf{K}\,I)(\mathbf{S}(\mathbf{K}\,I)\,I)$, where the two internal applications of the identity are "frozen" as arguments of $\mathbf{S}$. If this term is shared by other terms, the internal applications will be eventually duplicated: for this reason, the reduction of $(\underline{n}\underline{2}\,I\,a)$ is exponential in n, in Combinatory Logic (while we have seen that there exists a linear normalizing derivation for this term, and thus, a posteriori, a linear optimal reduction).

The previous examples are a consequence of the particular encoding of λ-abstraction into Combinatory Logic, but not of the reduction strategy. As a matter of fact, it is possible to prove that leftmost-outermost graph reduction of combinators is optimal. The problem is that optimality in Combinatory Logic has very little to do with optimality in λ-calculus.

Moreover, although one could consider different encodings of λ-abstraction, the problem above seems to be completely general: in any translation of a λ-term M into a CL-term M', there will be applications in M which are not translated as applications in M', but coded as arguments of some $\mathbf{S}$. When one of these $\mathbf{S}$ is shared, it must be duplicated, that implies a blind (not optimal) duplication of the corresponding λ-calculus application.

The situation is similar for SuperCombinators. Consider for instance the term ($\underline{2}\,\underline{2}\,\mathrm{I}\,a$), and note that Church integers are (a particular case of) supercombinators. The reduction of this term would proceed as in Figure 2.7.

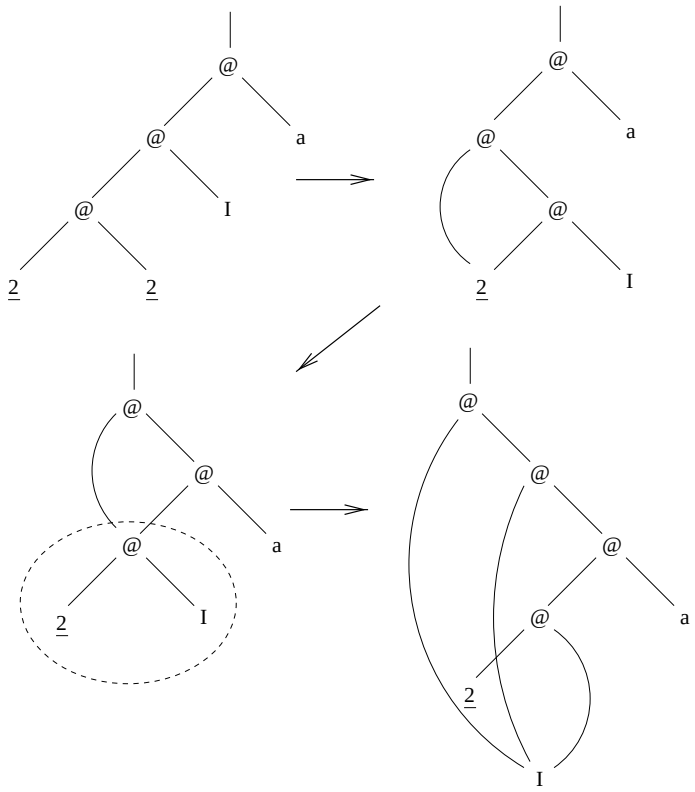


Fig. 2.7. SuperCombinator reduction

After two reduction steps, the subterm ($\underline{2}\,\mathrm{I}$) is shared (see the dotted line in Figure 2.7). Unfortunately, SuperCombinator reduction is not able to reduce this subterm to the identity ($\underline{2}$ does not have enough arguments: I is its only argument, in the subterm). So, we shall have two different reductions of this subterm, leading to a useless duplication of work. These duplications can combine in a very nasty way: it is easy to see that our typical example ($\underline{n}\,\underline{2}\,\mathrm{I}\,a$) is exponential in n, also in the case of SuperCombinators.

2.1.3 Environment Machines

An alternative way for reducing λ-terms that looks particularly promising in view of its sharing possibilities is by means of *environments*. Roughly, an environment is a collection of bindings, that is a mapping from variable names to values (other terms). Reducing a β-redex $\lambda x.(M\,N)$, the instance of the function M is not explicitly created, it is rather implicitly represented by means of a *closure* $\langle M, e\rangle$ between the body M and its current environment updated by the new binding $[N/x]$. If the function $\lambda x.M$ was shared, we may avoid duplicating the function: the difference between the instances of M will be expressed by their different closing environments. In particular, the body M, and hence any redex contained therein, have the potential to be shared in different closures, avoiding redundant reductions.

A number of reduction schemes for the λ-calculus using environments have been proposed in literature (e.g., the SECD machine by Landin [Lan63], the Categorical Abstract Machine by Cousineau, Curien and Mauny [CCM85], the K-machine by Krivine [], the λσ-calculus of Abadi, Cardelli, Curien, and Lévy [ACCL91, ACCL90]). A common framework for studying all these implementation is offered by the calculus of explicit substitution or λσ-calculus. The main idea of this calculus is to incorporate the meta-operation of substitution as an explicit component of the calculus. In particular, substitutions (environments) become first class objects in λσ-calculus. Working in this setting, it is possible to give a formal proof that even the sophisticated notion of sharing offered by closures is not enough to achieve optimal reductions. The following example is discussed by Field in [Fie90]:

$$M = (\lambda x.((x\,A)(x\,B))\;\lambda y.(\lambda z.((z\,C)(x\,D))\;\lambda w.(y\,E)))$$

(note the analogy with Lévy's example). Here, A and B are λ-abstractions, while C, D and E are arbitrary terms. After a few reductions, we *eventually* come to the following configuration

$$N = (\langle P, [A/y]\rangle\,\langle P, [B/y]\rangle)$$

where P is the *shared* closure

$$P = \langle((z\,C)(z\,D)), [\lambda w.(y\,E)/z]\rangle$$

Let us respectively call P_1 and P_2 the two external closures. Since all the terms in the closures of N are in normal form, we must proceed by applying some substitutions (i.e., computing some closures). If we start

reducing P, we eventually create two copies of the application $(A\,E)$ (and two copies of the application $(B\,E)$). These duplications are avoided if we start reducing P_1 and P_2, that is substituting A (respectively B) for y in $\lambda w.(y\,E)$. In this way, we get:

$$(((\lambda w.Q\,C)\,(\lambda w.Q\,D))\,((\lambda w.R\,C)\,(\lambda w.R\,D)))$$

where Q and R are the *shared* subterms $(A\,E)$ and $(B\,E)$. But, in this case we duplicate the redexes $\lambda w.((\ldots)\,C)$ and $\lambda w.((\ldots)\,D)$.

In conclusion, no matter what reduction is done first, some duplication of work eventually occurs.

Again, the example above may look artificial, but it should be correctly understood: its aim is to prove that shared environment and closure are not sufficient to implement optimal reduction *no matter what reduction strategy is adopted*. On the contrary, once a reduction strategy has been given, it is usually much simpler to find a counterexample to optimality. For instance, if we forbid reduction inside a λ, that is the typical weak notion of value adopted by most implementations, our usual term $(\underline{n}\,\underline{2}\,I\,a)$ is enough: its reduction turns out to be exponential in any environment machine (either strict or lazy).

2.2 Sharing graphs

The *initial* representation of a λ-term in the optimal graph reduction technique is very similar to its abstract syntax tree (just like in Wadsworth's graph reduction). There are however two main differences:

(i) we shall introduce an explicit node for sharing;
(ii) we shall suppose that variables are explicitly connected to their respective binders.

For instance, the graph in Figure 2.8 is the initial representation of the λ-term $M = (\delta\,\lambda h.(\delta\,(h\,I)))$, where $\delta = \lambda x.(x\,x)$ and $I = \lambda x.x$.

The triangle (we shall call it *fan*) is used to express the sharing between distinct occurrences of a same variable. All variables are connected to their respective binders (we shall always represent λ nodes drawing the binding connection on the left of the connection to the body). Since multiple occurrences of a same variable are shared by fans, we shall have a single edge leaving a λ towards its variables. So, each node in the graph (@, λ and fan) has exactly three distinguished sites (ports) where it will be connected to other ports.

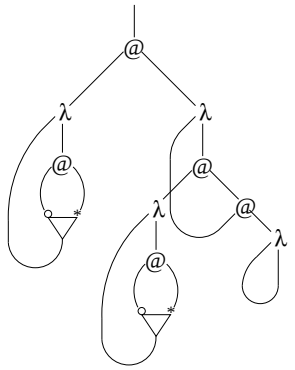


Fig. 2.8. Representation of $(\delta\ \lambda h.(\delta\ (h\ I)))$

2.2.1 Graph rewriting

We shall illustrate the main ideas of Lamping's optimal graph reduction technique showing how a simplified version of the algorithm would work on the λ-term $(\delta\ \lambda h.(\delta\ (h\ I)))$. As we shall see, a crucial issue will remain unresolved; we shall postpone its solution until Chapter 3.

The term $(\delta\ \lambda h.(\delta\ (h\ I)))$ reduces to the identity. The shortest reduction sequence consists in firing two outermost redexes first, and then proceed by an innermost reduction strategy (we leave the easy check of this fact to the reader):

$$\begin{aligned}
(\delta\ \lambda h.(\delta\ (h\ I))) &\rightarrow (\lambda h.(\delta\ (h\ I))\ \lambda h.(\delta\ (h\ I))) \\
&\rightarrow (\delta(\lambda h.(\delta\ (h\ I)))\ I) \\
&\rightarrow (\delta(\delta\ (I\ I))) \\
&\rightarrow (\delta(\delta\ I)) \\
&\rightarrow (\delta(I\ I)) \\
&\rightarrow (\delta\ I) \\
&\rightarrow (I\ I) \\
&\rightarrow I
\end{aligned}$$

In particular, we need 8 β-reduction steps. Note moreover that the inner redex $(\delta\ (h\ I))$ is duplicated by the first reduction.

The algorithm for optimal graph reduction consists of a set of local graph rewriting rules. At a given stage of the computation, we can usually have several reducible configurations in the graph. In this case, the choice of the next rule to apply is made non-deterministically. This

does not matter that much. As we shall see, the graph rewriting satisfies a one-step diamond property implying, not only confluence, but also that all the reduction sequences to the normal form (if it exists) have the same length. In our example, we shall usually choose the next rule according to a didactic criterion (and sometimes for graphical convenience).

By the way, the most important sharing graph rewriting rule is β-reduction: $\lambda x.(M\,N) \to M[N/x]$.

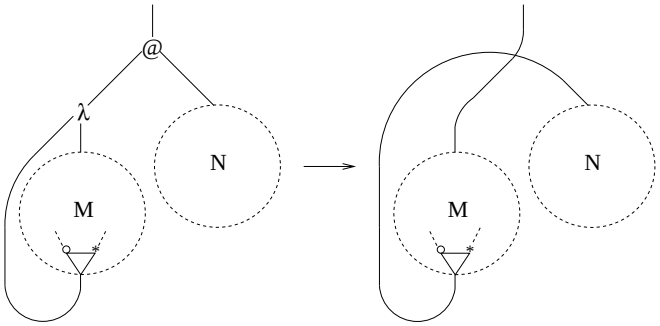


Fig. 2.9. β-reduction

In sharing graph reduction, substituting a variable x for a term N amounts to explicitly connect the variable to the term N. At the same time, the value returned by the application before firing the redex (the link above the application) becomes the instantiated body of the function (see Figure 2.9).

The portions of graph representing M and N do not play any role in the sharing graph β-rule. In other words, β-reduction is expressed by the completely local graph rewriting rule of Figure 2.10.

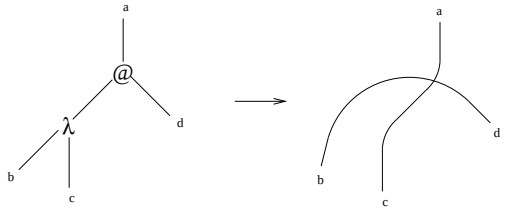


Fig. 2.10. β-rule

The term $(\delta\ \lambda h.(\delta\,(h\,I)))$ contains two β-redexes. The corresponding sharing graph reduction is given in Figure 2.11.

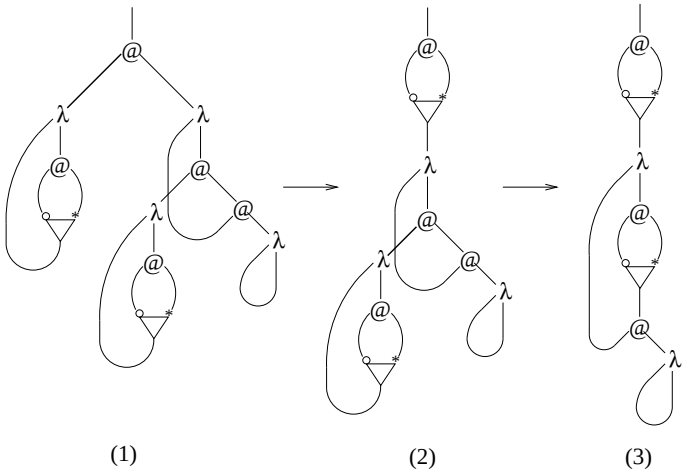


Fig. 2.11. Initial reductions of $(\delta\ \lambda h.(\delta\,(h\,I)))$.

The graph in Figure 2.11(3) corresponds to a shared representation of the λ-term

$$(\lambda h.((h\,I)\,(h\,I))\ \lambda h.((h\,I)\,(h\,I)))$$

in which the subexpression $(h\,I)$ is shared four times. Since the next redex involves a shared λ-expression, we must eventually proceed to the duplication of $\lambda h.(\ldots)$. In ordinary graph reduction, this duplication would be performed as a unique global step on the shared piece of graph. On the contrary, in the optimal graph reduction technique we proceed duplicating the external λ, and still sharing its body, as described in Figure 2.12.

Note that, duplicating the binder, we have been forced to introduce another fan on the edge leading from the binder to the variable: the occurrences of the variable are split in two, one for each new binder. In a sense, this upside down fan works as an "unsharing" operator (fan-out), that is to be "paired" against the fan(-in) sharing the body of the function. Although there is no operational distinction between a fan-in and a fan-out, their intuitive semantics is quite different; in particular, keep in mind that a fan-out is always supposed to be paired with some fan-in in the graph, delimiting its scope and annihilating its sharing effect. As we shall see, the way in which the correct pairing between fans is determined is a crucial point of the optimal graph reduction

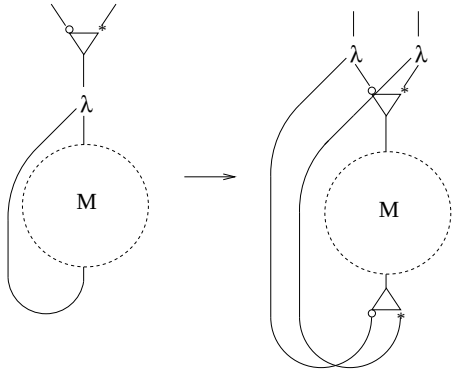


Fig. 2.12. The "duplication" of λx.M

technique. But for the moment, we leave this point to the intuition of the reader.

Again, the body of the function λx.M does not play any role in Figure 2.12. The reduction rule can thus be formally expressed as a local interaction between a fan and a λ, as described in Figure 2.13.

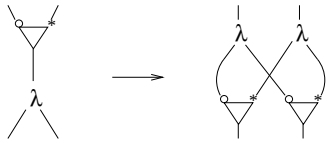


Fig. 2.13. Fan-λ interaction

Let us proceed reducing our example. After the application of the fan-λ interaction rule, we get the graph in Figure 2.14(4). A new β-redex has been created, and its firing leads to the graph in Figure 2.14(5), corresponding to the λ-term

$$((\lambda h.((h\,I)\,(h\,I))\,I)\,(\lambda h.((h\,I)\,(h\,I))\,I))$$

In the graph, there are no more β-redexes and no fan-λ interactions. Therefore, we must proceed duplicating some application. If we would duplicate the outermost application, we eventually loose sharing: it would amount to duplicate the redex $(\lambda h.((h\,I)\,(h\,I))\,I)$. Therefore, in general, the graph rewriting rule of Figure 2.15 is strictly forbidden, in the optimal implementation technique (although semantically correct). The intuition should be clear: since the shared application could be in-

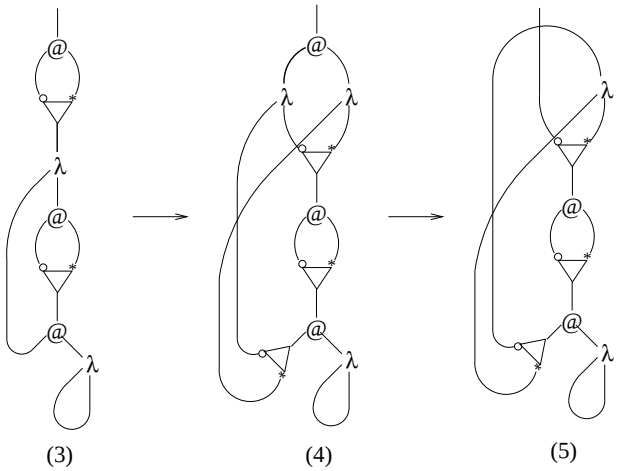


Fig. 2.14. ... reduction ...

volved in some redex, its duplication would imply a double execution of the redex.

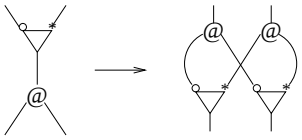


Fig. 2.15. Non optimal duplication

Let us then consider the innermost application. We already know that we cannot apply the rule of Figure 2.15. However, in this case, we have another fan (a fan-out) that could possibly interact with the application, via the obvious rule of Figure 2.16. We have to understand if such a duplication preserves optimality.

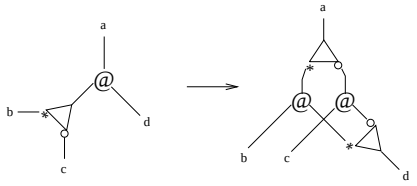


Fig. 2.16. Fan-@ interaction

Note that the innermost application corresponds to six applications in the associated λ-term: two applications of the kind $(\lambda h.((h\,I)\,(h\,I))\,I)$, and four of the kind (hI). The first group corresponds to the path leading from the application to $\lambda h.(\ldots)$ passing through the port marked with $\circ$ of the fan-out; similarly, the other group corresponds to the path leading from the application to the binding port of the λ, passing through the port marked with $*$. There is no intuitive way to handle these two groups of reductions together, since we do not know anything about the term that will be substituted for h. In general, from the point of view of sharing, it is always correct to proceed to the duplication of the application according to Figure 2.16, because such a configuration already implies the existence of two unsharable (class of) redexes for the application.

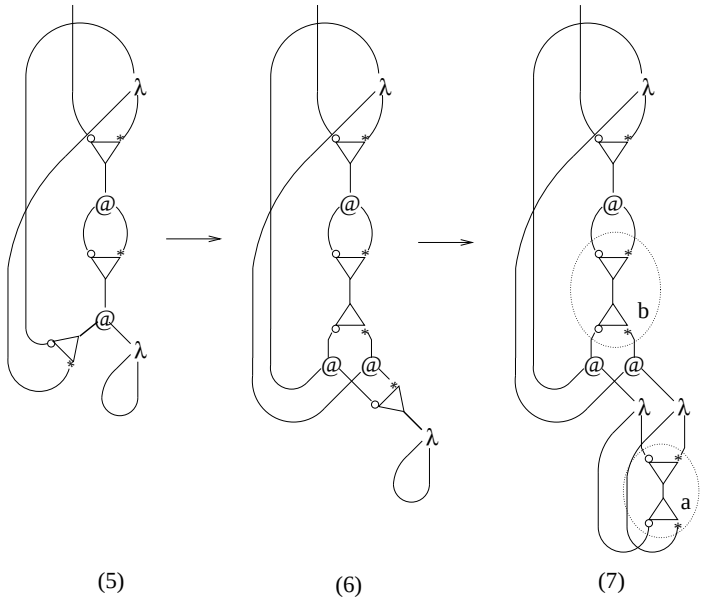


Fig. 2.17. fan-@ interaction

The reduction of our example proceeds according to Figure 2.17. We fire the fan-@ interaction rule, putting in evidence a new fan-λ rule. In the final graph, we have two groups of fans meeting each other (see the dotted ovals a and b in the picture). This is another crucial point of the optimal graph reduction technique. As we shall see, these two cases

must be handled in different ways (although, up to now, we have no syntactical way to distinguish them).

Let us start with the interaction marked with a in Figure 2.17. We are duplicating the identity, and the obvious idea would be to complete the duplication. This amounts to annihilate both fans connecting the corresponding links: $\circ$ with $\circ$, and $*$ with $*$, see Figure 2.18(1).

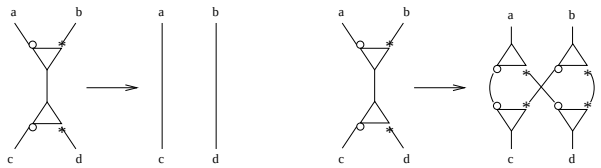


Fig. 2.18. Fan-fan interactions.

This rule can be only used when the fan-in and the fan-out are paired (since, in a sense, they belong to a same "duplication process" that has been now accomplished). This is not the case in the configuration marked with b in Figure 2.17, where the fan-in and the fan-out have nothing to do with each other (the fan-out is actually paired with the uppermost fan-in). In this case, the fans must duplicate each other, according to the rule in Figure 2.18(2). The problem of deciding which rule to apply when two fans meet (that is the question of how their pairing is established) is a crucial point of the optimal implementation technique. But as previously said, we shall postpone this complex matter to a further chapter; for the moment, we will use intuition (the informal arguments presented so far) to decide whether two fans pair or not.

Applying the fan-fan interaction rules of Figure 2.18, the computation proceed as described in Figure 2.19.

The rules seen so far are enough to complete the reduction. In particular, we can start firing two β-redexes, we can then duplicate the only application left in the graph and annihilate a couple of paired fans.

The final graph in Figure 2.20 has been redrawn for the sake of clarity in Figure 2.21. Now, the identity gets duplicated, its application is reduced. The resulting identity id duplicated again and by a final β-redex we get the expected normal form.

The normalization of this term required only 7 applications of the β-rule, against 8 β-reductions needed by the *best strategy* in the usual syntactical universe of λ-expressions (recall that there is no effective way

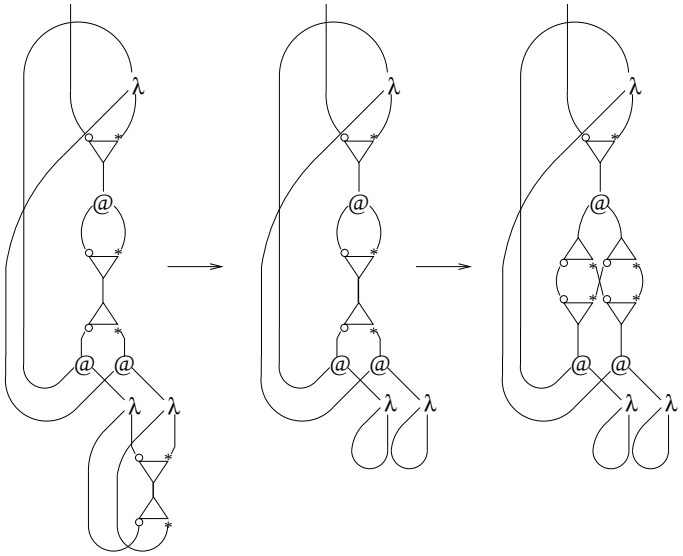


Fig. 2.19. ... reduction ...

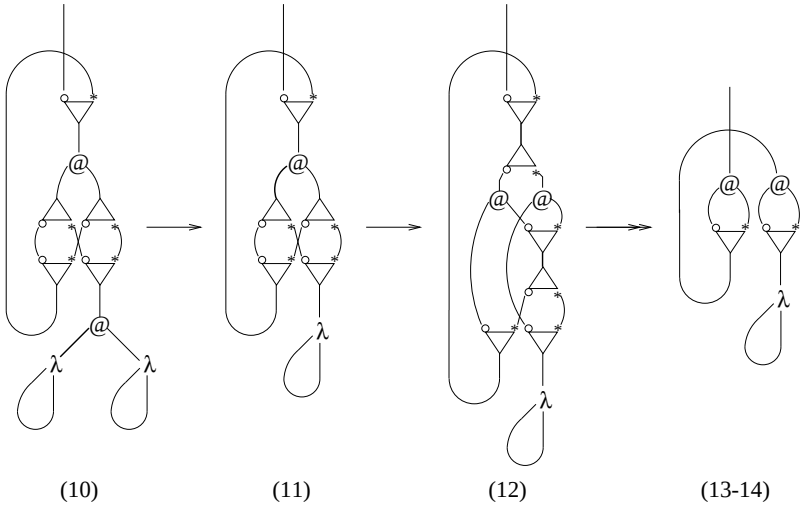


Fig. 2.20. ... reduction ...

to choose the best reduction order in the λ-calculus). The situation is

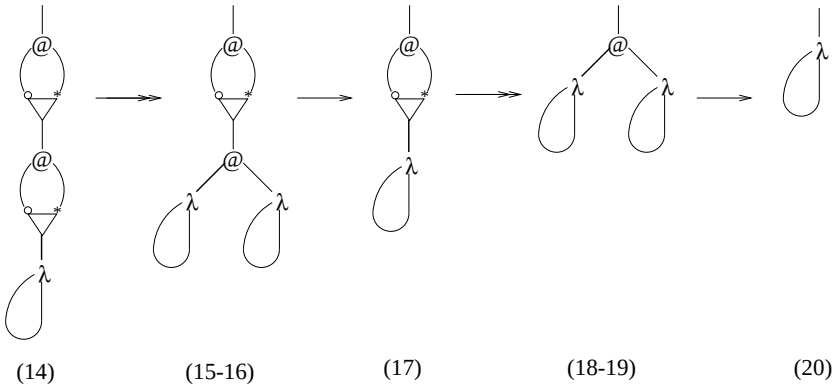


Fig. 2.21. ... reduction.

even more favorable for other typical reduction strategies: innermost and outermost reduction require respectively 12 and 14 β-reduction steps. Although this is still not very impressive, the actual gain of the optimal graph reduction technique can become exponential on more complex examples.

Let us finally recall, for the sake of clarity, all the graph reduction rules considered so far, see Figure 2.22. We stress again that the main problem of the optimal graph reduction technique consists in discriminating the application of the last two rules in Figure 2.22. By the previous discussion, this amounts to understand how fan-in and fan-out are paired in the graph. We suggest the reader to try to think of this problem by himself, before proceeding any further in reading the book.

Let us also remark that all graph rewriting rules in Figure 2.22 can be seen as local interactions between pair of nodes in the graph. Such graph rewriting systems have been studied by Lafont, under the name of Interaction Nets. Note in particular that each node has a unique, distinguished port where it can possibly interact with other nodes. This port, that has been put in evidence by an arrow in Figure 2.23, will be called the *principal port* of the node. All the other ports will be called the *auxiliary port* of the node.

2.2.2 Read-back and Semantics

The initial sharing graph representation $\mathcal{G}_M$ of a λ-term M has an obvious correspondence with the abstract syntax tree of M. Besides, as soon

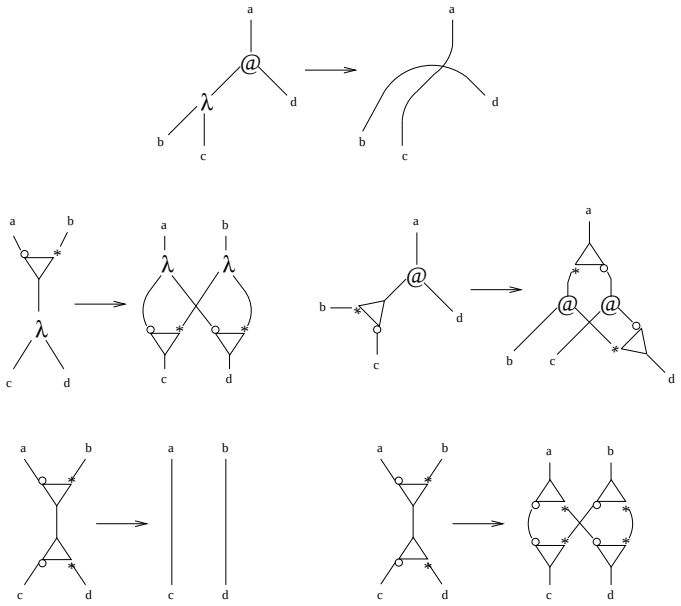


Fig. 2.22. Sharing graph reduction rules.

Fig. 2.23. Principal ports.

as we start the reduction (and we introduce fan-out), this property is immediately loss, and quite complex loops arise in the graph. Moreover, in general the normal graph $\mathcal{G}'$ of $\mathcal{G}_M$ is quite different from the graph $\mathcal{G}_N$ encoding the normal form N of M.

Consider for instance the λ-term

$$M = \lambda x \lambda y.(\lambda z.(z\,(z\,y))\;\lambda w.(x\,w))$$

This term reduces to the Church integer $\underline{2} = \lambda x \lambda y.(x\,(x\,y))$. Nevertheless, its graph reduction (see Figure 2.25) produces the normal form in Figure 2.24, that is not the initial encoding of $\underline{2}$.

The obvious problem is to understand in which sense a sharing graph "represents" a λ-term, or, if you prefer, how can we "read-back" a λ-expression from a sharing graph (in some sense, sharing graphs should be considered a kind of "internal representation" of the associated λ-

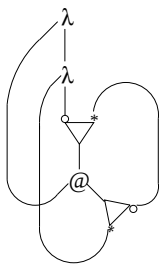


Fig. 2.24. A shared representation of Church's $\underline{2}$.

expression). This problem is indeed essential in order to prove the correctness of the algorithm: since we *do not have* syntactical correctness (the normal forms do not coincide), we must eventually work up to some semantical notion of equivalence between sharing graphs†.

The general idea behind the read-back is to compute the abstract syntax tree of the corresponding λ-term by considering one branch at a time. Although the graphs we considered so far are not directed, we have already remarked that it looks natural to superimpose a top-down direction (that, informally, also allowed us to distinguish a fan-in from a fan-out)†. The idea is to read each branch of the syntax tree represented by the graph by following a path from the root node up to a bound variable, traveling top-down in the graph. Note that, when we reach a fan-in, we eventually enter this form at an auxiliary port, and we exit from its principal port. Conversely, when we reach a fan-out, we forcedly enter through its principal port, and we must decide from which auxiliary port to exit: from the one marked with $\circ$ or from the one marked with $\star$. In order to solve this ambiguity, recall that each fan-out is always "paired" with some fan-in, and that the fan-out is supposed to annihilate the sharing effect of the paired fan-in. It is possible to prove that, reading paths in the way mentioned above, the fan-in paired with some fan-out has been eventually traversed in the path leading from the root to the fan-out. Now, the decision about which way to follow exiting from a fan-out is obviously imposed by the port we used to enter the

† The read-back problem is pretty semantical: operationally, we do not usually need to read-back the result, in the same way as we never read-back the internal representation of a function in all the usual implementations of functional programming languages. If the result is a basic value, it will be directly available as root node at the end of the computation.

† This property could be formalized by introducing a suitable polarity for all the ports of the forms in the graph.

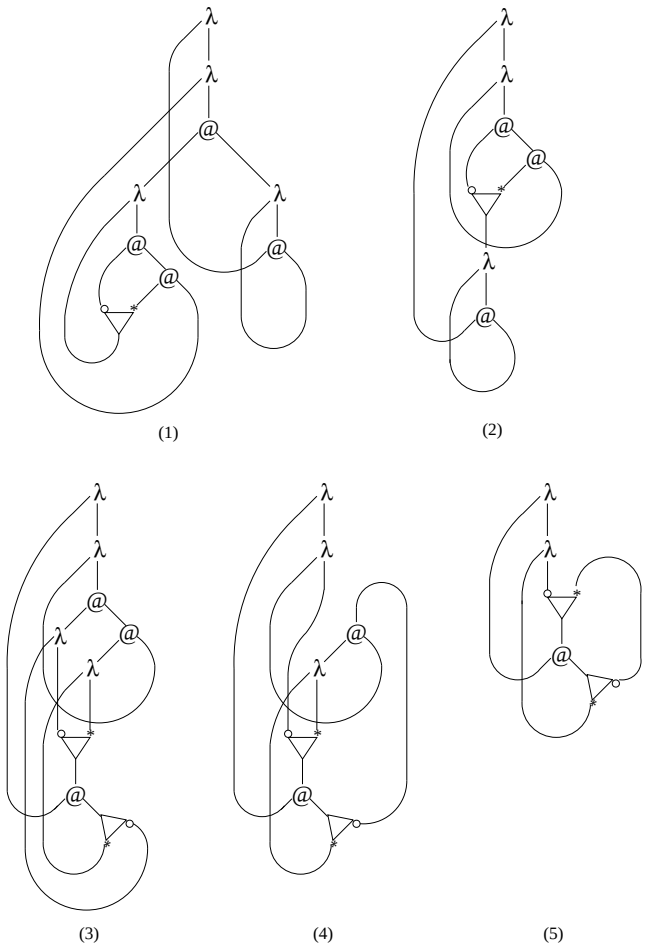


Fig. 2.25. Sharing reduction of $M = \lambda x \lambda y.(\lambda z.(z\,(z\,y))\;\lambda w.(x\,w))$

.

paired fan-in: if we entered the fan-in at $\circ$, we shall exit the fan-out at $\circ$, and similarly for $\star$.

Let us see how the read-back algorithm would work on the example in Figure 2.24. Starting form the root, we traverse two λ (we always exit towards the right-auxiliary port, corresponding to the body). Then we reach a fan-in, and an application node. Now we are free to choose which branch (which argument of the application) reading first. Let us suppose to exit towards the functional argument (that is, exiting from

the principal port of the abstraction node). We find a loop towards a λ node, corresponding to a bound variable (moreover, we know that the variable was bound by the first binder we traversed). So, we have finished, along this path. The portion of term read-back so far is drawn in Figure 2.26(a). Let us start the process again, but this time choosing to exit from the auxiliary port of the application. We reach a fan-out. Since we entered the paired fan-in at the port marked with ∘, we are forced to exit from the corresponding port of the fan-out. We traverse a second time the fan-in (this time entering from the port marked with ⋆), and we find again the same application as before. Obviously, choosing to follow the path towards the function, we find the same bound variable of the first time (see Figure 2.26(b)).

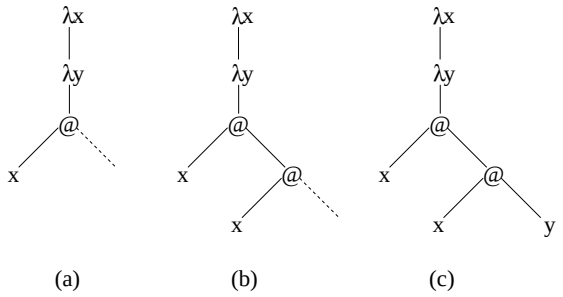


Fig. 2.26. Read-back.

To conclude, let us resume our read-back process starting from the auxiliary port of the last application. We meet again the fan-out, but this time we are forced to exit from the port marked with ⋆, since this was the port through which we entered the last occurrence of the paired fan-in. We find a variable bound by the second `lambda` (see Figure 2.26(c)). Since all the paths have been considered, we are done.

3
The full algorithm

The main problem of the Optimal Reduction Algorithm via Sharing Graphs is to establish the correct pairing between fan-in's and fan-out's. The most obvious idea would be to label fans in the sharing graph, in such a way that two fans are paired if and only if they have identical labels. Unfortunately, this simple solution is inadequate. A nice argumentation against labels was given by Lamping in [Lam89].

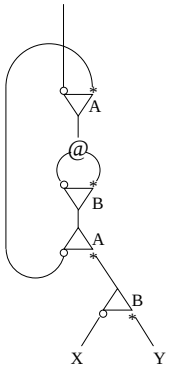


Fig. 3.1. Ambiguous pairing of fans.

As we have seen in the introduction, complex looping paths are created along the reduction of a sharing graph. This looping paths present a problem with labeling when a path traverses several times the same fan-in before reaching its paired fan-out. For instance, in the graph of Figure 3.1, suppose that the fans are paired according to their labels. Any legal path that reaches the fan-out labeled B must eventually traverse the fan-in labeled B twice. If the traversal of the fan-out is paired

with the first traversal of the fan-in, the graph represents the expression $((x\ x)\ (y\ y))$; on the contrary, if the second traversal of the fan-in B is taken into account, the read-back of the graph is $((x\ y)\ (x\ y))$. We can then conclude that labels are not enough to distinguish these two cases.

Exercise 3.0.1 Prove that the graph of Figure 3.1 can respectively occur with the two interpretations above near the end of the reduction of the following terms:

(i)

$$\begin{array}{l}\lambda x.\lambda y.\ (\ \lambda f.(\ \lambda h.\ (h\ \lambda p.\ (h\ \lambda q.\ p))\\ \qquad\qquad\qquad \lambda l.\ (((f\ \lambda n.\ (l\ n))\ x)\ y)\ )\\ \qquad\qquad\quad \lambda g.\lambda u.\lambda v.((g\ u)\ (g\ v))\ )\end{array}$$

(ii)

$$\begin{array}{l}\lambda x.\lambda y.\ (\ \lambda f.(\ \lambda h.\ (h\ \lambda p.\ (h\ \lambda q.\ q))\\ \qquad\qquad\qquad \lambda l.\ (((f\ \lambda n.\ (l\ n))\ x)\ y)\ )\\ \qquad\qquad\quad \lambda g.\lambda u.\lambda v.((g\ u)\ (g\ v))\ )\end{array}$$

The next example provides a more operational grasp of the problems with labeling. In Figure 3.2 is described the initial part of the sharing graph reduction of $(\delta\ \delta)$. In the initial graph, the two fans are not paired, so they get different labels (A and B, in our example). The label of the fan marked with A must be preserved while it traverses the λ in the second reduction (the two "residual" fans are actually paired). The problem, is the mutual crossing of fans, in the third reduction. If labels are unstructured, this rule must be symmetric. Since the fan-out marked with A is paired with the top-most fan-in (also marked with A), this label must be preserved during the mutual traversal of fans. Due to the symmetry of the rule, this implies that the labels of fans must be left unchanged during their mutual traversal; in particular, we create two residual fan-in's both marked with B. The problem is that these two fans have nothing to do with each other. As a consequence, at the end of the reduction represented in Figure 3.2, we (correctly) get a graph with the same shape of the initial one, but where the two fan-in's look paired. This would eventually turn into an error in continuing the reduction.

Exercise 3.0.2 Continue the reduction of the example in Figure 3.2.

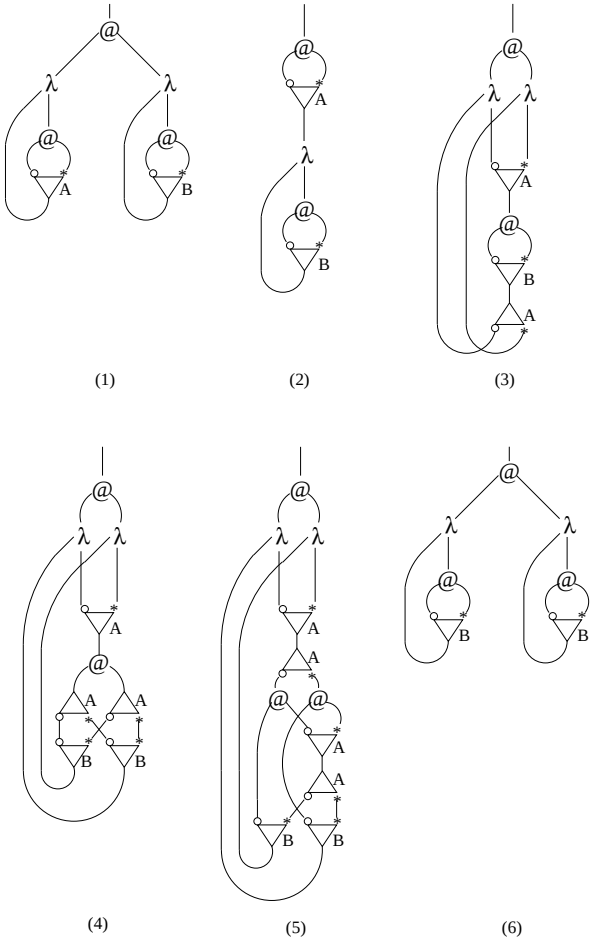


Fig. 3.2. Labeling fans with names.

3.1 Pairing fans

Since labels are not enough to solve the problem of pairing fans, we should consider more structured decorations of the nodes in the graph. Looking back at the mutual crossing rule for fans, one can easily realize that, semantically, this rule is not as symmetric as it looks like: the idea is that, when two fans meet in such a configuration, they are not mutually duplicating, there is instead only one "active" fan which is duplicating (and passing through) the other. Starting from this simple

consideration, we can imagine that we need some order relation among labels.

Let us just take integers. In other words, let us add a local level structure to the bidimensional graphs presented in the introduction. Each operator will be decorated with an integer tag which specifies the level at which the operator lives: two fans match, or are paired, if they meet at the same level; they mismatch otherwise.

Intuitively, one may think of the level of a node in the graph as expressing the number of different ways the fan can be shared inside the term. Consider for instance the term (P (Q R)). The subterm (Q R) can become shared inside P; since we must avoid any conflict between the possible duplication of (Q R) and the fans internal to this term, (Q R) must be put at a different (let us say, higher) level than P. Similarly, R is not only potentially shared by P as a subterm of (Q R), but it can be also shared inside Q. So, it must live at an even higher level than Q. More generally, every time we pass the argument of an application, we should increment the level of the term. For instance, the graph in Figure 3.3 is the already met term (δ δ) with the right node indexing.

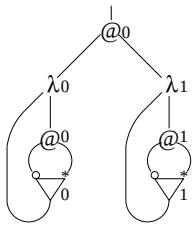


Fig. 3.3. Syntax tree of (δ δ)—Adding levels.

Clearly, this is not enough to solve our problems: in the same way as for labels, after one iteration of the reduction process in Figure 3.2, the whole term would be at level 1, while one would expect to get the same configuration of Figure 3.3. We need therefore a further operator for decreasing the level of nodes. This operator is usually called "croissant", due to its customary representation, see Figure 3.4.

Fig. 3.4. The "croissant" operator.

The idea is that a croissant travels along the other nodes of the graph in the direction of the arrow decrementing the level of the operators that it crosses (see the left-hand side rule of Figure 3.5). When two croissants meet face to face at the same level, they are annihilated, as in the case of fans (see the right-hand side rule of Figure 3.5).

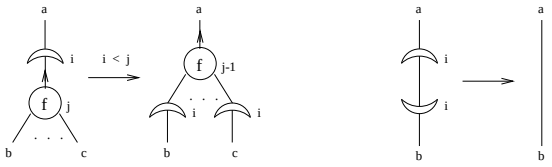


Fig. 3.5. Rewriting rules for "croissant".

The problem is where to put such operators in the initial graph. By the discussion above, this question can be rephrased in the following way: when does a potentially sharable object loose a level of sharing? Stated in this way, the answer is easy: when the object is accessed by an occurrence of a variable (occurrences of variables are linear entities). As a consequence, we should add a croissant every time we have an occurrence of a variable in the term. In the case of $(\delta\ \delta)$, we would get the graph of Figure 3.6.

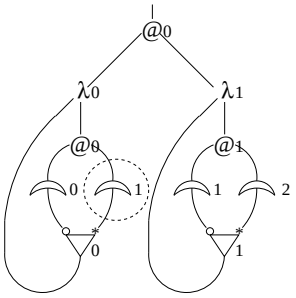


Fig. 3.6. Syntax tree of $(\delta\ \delta)$—Adding croissants.

This works fine for the leftmost copy of δ which is created along the reduction, since it is now shifted at level 0, as we expected. However, we have now problems with the other copy. The source of the problem should be clear, by now: since this copy is going to be accessed inside the argument of an application (see the dotted region in Figure 3.6), it should be incremented by one level before doing any use of it. To this

aim, we are forced to introduce yet another operator called "bracket" (see Figure 3.7).

Fig. 3.7. The "bracket" operator.

A bracket works much like a croissant, but it increments levels, instead of decrementing them. The corresponding rewriting rules are given in Figure 3.8.

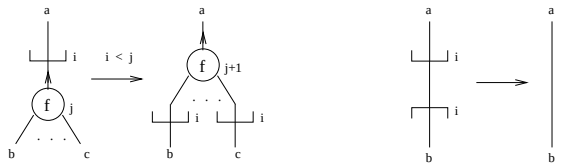


Fig. 3.8. Rewriting rules for "bracket".

In the initial graph, a bracket must be put in correspondence with every free variable inside an argument of an application.

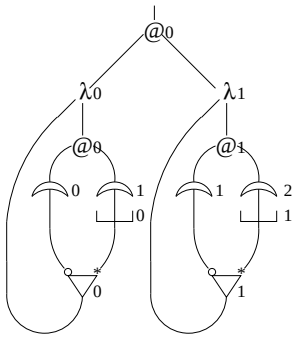


Fig. 3.9. The initial translation of (δ δ).

Putting all together, we finally get the correct translation of (δ δ) drawn in Figure 3.9. The formal statement of the inductive rules for the translation of λ-terms into sharing graphs can be found in section 3.3.

3.1.0.1 Some additional remarks on brackets and labels

The above introduction to sharing graphs and our solution to the problem of pairing is sensibly different from Lamping's original presentation. Lamping's idea was to introduce a notion of enclosure to delimit the interaction of fans, in such a way that fans from different enclosures never pair. Croissants and brackets are used to delimit these enclosures: in particular, in the initial graph, they just surround each fan in the graph. Unfortunately, this simple idea is not fully adequate: as Lamping observe (see [Lam89], p.29) "when initially encoding a λ-expression with a graph, a few extra brackets may be necessary ... This kind of bracketing ... is only needed to delimit the scope of a lambda that contains free variables". Actually, this is also the reason why Lamping did never provide a clear inductive definition of his translation of λ-expressions into sharing graphs. A more formal explanation of these extra-brackets was given in [GAL92b]. The nice idea was that application and abstraction nodes should be assimilated to fans (note indeed their similar operational behavior), requiring thus a suitable enclosure themselves.

The translation presented here was proposed for the first time in [Asp94]. The main advantage of this translation, as we shall see, is that of being particularly close to Linear Logic and Geometry of Interaction. Actually, the general intuition is not that far from Lamping: we still use croissants and brackets to define enclosures. However, we are not enclosing fans, but "sharable data" (boxes, in the terminology of Linear Logic). We shall come back to the other translations in Chapter 8

3.2 Sharing Graphs

As we have seen in the previous section, our original definition of sharing graph must be extended with two new operators required for operating on the level structure of the graph: the *croissant*, which opens or closes a level, and the *square bracket* (or simply *bracket*), which temporarily closes a level or restores a temporarily closed one. Summarizing, sharing graphs are undirected graphs built from the indexed nodes in Figure 3.10.

Each node of the graph has a *main* or *principal* port where it can possibly interact with other nodes (this port has been put in evidence by an arrow in Figure 3.10). All the other ports of the node are its *auxiliary* ports.

Two operators (nodes of the graph) are said to *interact* when their

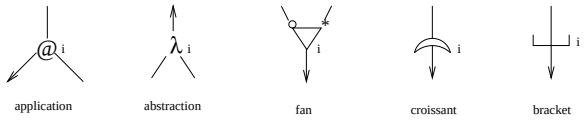


Fig. 3.10. Sharing graph operators.

principal ports are connected and their levels are equal. The whole set of the interaction rules introduced so far is described in Figure 3.11.

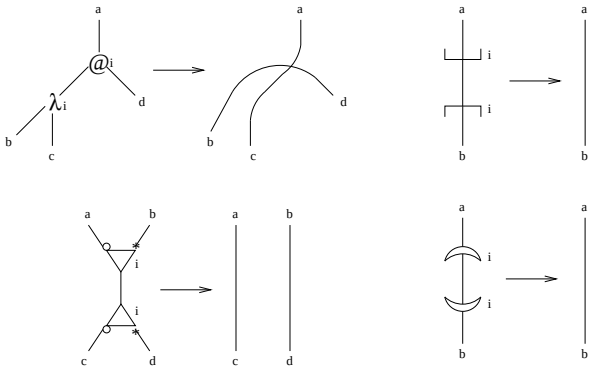


Fig. 3.11. Interaction rules.

The first of these rules is the usual β-reduction. The second one (bottom-left in figure) is the matching of fans. The latter two state respectively that opening and then immediately closing a level, or temporarily closing and then immediately reopening it has no effect at all.

An operator at a given level can also act upon any other operator `f` at a higher level (reached at its principal port), according to the rules in Figure 3.12 (where `f` denotes a generic operator).

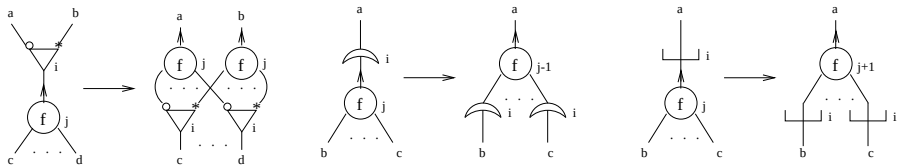


Fig. 3.12. Action rules.

In these rules, the operators are simply propagated through each other

in such a way that their effect on the level structure is left unchanged. The terminology is due to the following fact: the active operator can have an effect on the level at which the passive one is found, so the index of the latter can be changed, while the index of the former remains the same.

Although we made a "semantical" distinction between action and interaction, all of the graph rewriting rules are local interactions in Lafont's terminology. We shall often use the term interaction in this more generic sense.

3.3 The initial encoding of λ-terms

A λ-term N with n free variables will be represented by a graph with $n+1$ entries (free edges): n for the free variables (the inputs), and one for the "root" of the term (the output). The translation is inductively defined by the rules in Figure 3.13.

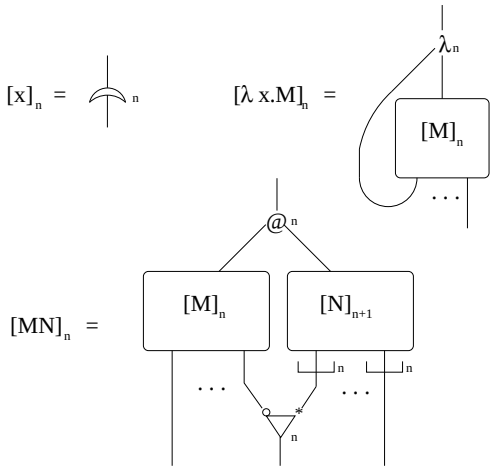


Fig. 3.13. Initial translation.

The translation function is indexed by an integer which can be thought of as being the level at which we want the root to be. The translation starts at level 0, *i.e.* $[M] = [M]_0$.

In the figure for the application (M N), all the variables common to M and N are supposed to be shared by means of fans. As a consequence, we always have just one entry for each distinct variable in the graph.

If a lambda abstraction binds a variable that does not occur (i.e., it is free) in its body, the edge of the lambda node leading to the variable would be dangling. To avoid this dangling reference, we introduce a new operator, usually called *garbage*. As far as we are not concerned with garbage collection issues, this operator has no operational effect in the reduction system.

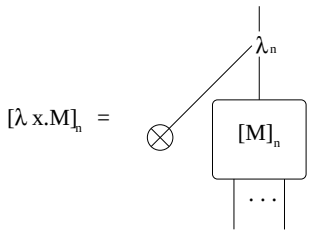


Fig. 3.14. The translation of $\lambda x.M$ when $x \notin FV(M)$.

Example 3.3.1 Figure 3.15 shows the initial encoding of the λ-term ($\underline{2}$ I).

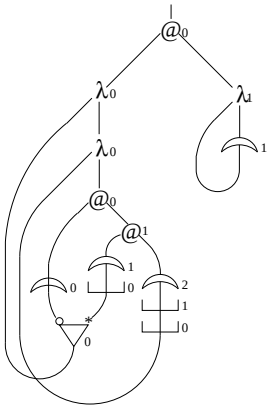


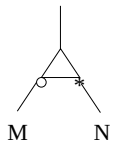
Fig. 3.15. Initial translation of ($\underline{2}$ I).

Exercise 3.3.2 Translate the terms of Exercise 3.0.1 applying the rules in Figure 3.13. Use the sharing graph reduction rules so far defined to reduce the sharing graphs obtained by the translation. Remark the

indexes of the fans in the configurations analogue to the one of Figure 3.1 that would be reach near the end of the reduction.

3.4 Lamping's paths

The correctness of sharing implementation is proved by defining a matching between sharing graphs and terms. We have already remarked that there is a close correspondence between the syntax tree of a λ-term M and its sharing graph representation [M], and that this correspondence is far from evident as soon as [M] is evaluated (see Figure 2.24, for instance). The crucial point is the presence of subpieces of the graph which contain fan-out's.



For instance, the above subgraph may represent either M or N. This uncertainty may be resolved by observing that a fan-out describes "unsharing" and that an expression/subgraph which contains unsharing must have been shared somewhere "in the history of the path" reaching the node. Namely, if we take a *path* from the root of the graph to a fan-out node, we expect to find some fan-in performing the sharing paired with the fan-out. Once this pairing has been established, the branch for exiting the fan-out must be the same of the one that has been traversed at the paired fan-in. This means that we must record which branches of fan-in's have been crossed along the path. The structure where this information is stored is called *context*.

The branching information is still insufficient for recovering the proper pairing of fans. In fact, paths may traverse several fan-in's (even the same!) before finding a fan-out (see Figure 3.1). In order to overcome this problem, the mechanism of contexts must not only store the branching information of fan-in's, but also organize this information in order to determine the pairing between fans.

The level structure of nodes suggests that contexts too should have a similar structure. In this way, traversing a $*$-branch of an n-indexed fan-in means recording this information in the n-th level of the context. The traversal of an n-indexed fan-out is thus constrained by what is stored in the n-level of the context. More precisely, let a context be a list $\langle\langle\cdots\langle C, a_{n-1}\rangle,\cdots, a_1\rangle, a_0\rangle$ whose elements a_i may be lists at their

turn. The tail of A at width n is a context C, i.e., the context C is the information stored in A starting from level n. Since a node with index n operates on such a context C, we shall use $A^n[\cdot]$ to denote the context whose first n elements are the first n elements of A, while its tail at level n is unspecified, that is,

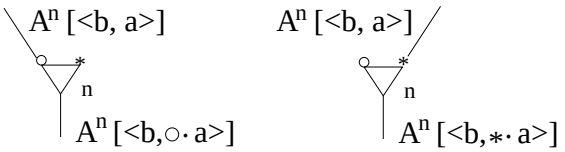
$$A^n[\cdot] = \langle\langle\cdots\langle\cdot, a_{n-1}\rangle,\cdots, a_1\rangle, a_0\rangle.$$

Inserting a context into $[\cdot]$ we get then

$$A^n[C] = \langle\langle\cdots\langle C, a_{n-1}\rangle,\cdots, a_1\rangle, a_0\rangle.$$

We stress that C can be any context. So, in general $A^n[C] = A$ if and only if C is the subcontext of A at width n.

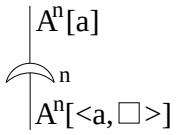
Using the latter notations, the context transformation performed by a fan-in may be summarized as follows:



Fan-out's transform contexts in a dual way with respect to fan-in's (turn upside-down the above picture). Namely, when a fan-out is met, we choose the $*$-branch or the $\circ$-branch for prosecuting the path, according to what is stored at level n in the context. Of course, after traversing the fan-out the topmost information at level n in the context is popped (since it has been consumed).

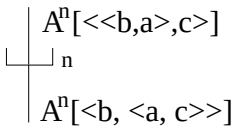
What about when a croissant or a bracket is met? Well, recall from section 3.1 that these nodes were introduced with the purpose of "managing the level of sharing". In particular, croissants allow to decrement the level of sharing of a graph. That is, an n-indexed croissant in front of a graph at level k, $k > n$, means that the graph must be read at level $k-1$. From the point of view of contexts this operation of "decreasing the level of sharing" is reflected by the operation which "removes one level" from the context, the n-th level in the above case. There is a problem: levels of contexts store the branch marks of the traversed fan-in's. If we remove this information, there is no way for restoring it when, later on the path, a paired croissant is found (and we have to "increase the level of sharing"). In other words we must understand whether the removed information at level n may be meaningful in the following of

the path or not. To this aim remark that n-indexed croissants instantiate variable occurrences with the associated expressions, which must be at a level greater than n. We expect that no branch information is recorded at the n-th level when the croissant is met, for every node of the associated expression stays at a greater level. Namely, the n-th level of the context must be empty (notation: $\Box$) when a croissant is met. The following picture describes this requirement:



Note that crossing a croissant top-down all the contexts at levels greater or equals than n increase their level by one. While on the contrary, going bottom-up the head of the context at level n is removed and all the following contexts shift to the right decreasing their level by 1.

Finally, let us come to brackets. We said that the traversal of brackets performs a "dual" action with respect to croissants. This is not completely true. Actually, the aim of brackets is to *define* the part of the graph which must be shared. Of course the sharing level increases when you enter this part. But, unlike the case of croissants, we cannot hope to enter an n-indexed bracket with an "empty" level n of the context. Indeed, when you exit a "shared part" at level n, we might have recorded branching informations required in the following, accessing again the shared part. This means that the information at level n must be saved when the shared part is left and resumed when the shared part is entered again. In terms of contexts we need an operation that save the information at a given level. We shall use the same operator for building levels, as drawn in the following picture:



Remark that, as we have already seen for the croissant, also the bracket changes the level of the contexts above n. We also invite the reader to check that the opposite rewriting behavior of bracket and croissant is sound with respect to their corresponding operations on contexts.

The operational behavior of bracket and croissant is reversible. In particular, traversing a bracket top-down (according to the previous

picture) we take the context $\mathfrak{a}$ at level $\mathfrak{n}+1$ and we save it into the context at level $\mathfrak{n}$ creating a pair $\langle \mathfrak{a}, \mathfrak{c} \rangle$, so that no information would be lost. Vice versa, traversing a matching bracket the pair $\langle \mathfrak{a}, \mathfrak{c} \rangle$ is split and the contexts $\mathfrak{a}$ and $\mathfrak{c}$ become available again. The last point is one of the most important. It means that in order to access the information we saved crossing a bracket top-down (such a direction correspond to the fan-in direction in the case of fans) we need to traverse bottom-up (always with respect to the previous picture) a bracket paired with the previous one. As a consequence, no fan, bracket or croissant paired with a node that operated on $\mathfrak{a}$ or $\mathfrak{c}$ can be met in between.

It remains to deal with abstraction and application nodes. Does these nodes modify contexts? Lamping, when he proved the correctness of his implementation [Lam89], did not define context transformations for @ and λ-nodes. Indeed, in order to read-back the λ-term associated to a sharing graph (which was Lamping's purpose) it is enough to recognize those paths of the sharing graph which are in one-to-one correspondence with the paths in the syntax tree of the term. To this aim, when an @-node is met you must prosecute both from the function port and from the argument port (you have to read-back both the arguments of the application); when a λ is met you must prosecute from the body port (you must read-back the body). In other words, the node itself specifies how to lengthen the path and there is no necessity for context information.

As far as this chapter is concerned, Lamping's paths work very well (we want to define the λ-term associated to a sharing graph, too). However, we warn the reader that the notion of proper path we are using here does not fit with the notion of path we will use in Chapter 6. There, we shall come back in more detail on this issue.

3.4.1 Contexts and proper paths

Let us formalize the foregoing discussion. Foremost we shall define contexts.

Definition 3.4.1 (context) A *level* is an element of the set inductively generated by the following grammar:

(i) $\Box$ is a level (the *empty* level);

(ii) if $\mathfrak{a}$ is a level then so are $\circ \cdot \mathfrak{a}$, $* \cdot \mathfrak{a}$;

(iii) if $\mathfrak{a}$ and $\mathfrak{b}$ are levels, then also $\langle \mathfrak{a}, \mathfrak{b} \rangle$ is a level.

A *context* is an infinite list of levels containing only a finite number of non-empty levels. Namely:

(i) the *empty* context $\varnothing$ is the infinite list of empty levels, i.e., $\varnothing = \langle \varnothing, \Box \rangle$;

(ii) if C is a context and $\mathfrak{a}_0$ is a level, then $\langle \mathsf{C}, \mathfrak{a}_0 \rangle$ is a context.

According to the previous definition, a context has the following shape:

$$\mathsf{A} = \langle \cdots \langle \mathsf{B}, \mathfrak{a}_{n-1} \rangle \cdots, \mathfrak{a}_0 \rangle$$

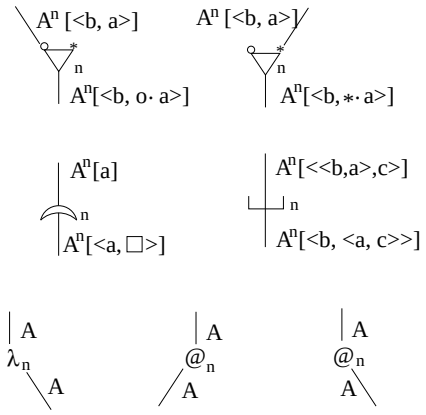
where $\mathfrak{a}_0, \ldots, \mathfrak{a}_{n-1}$ are levels, and B is a context. We will say that $\mathsf{A}(i) = \mathfrak{a}_i$ is the i-th level of the context A, and that B is the subcontext of A at *width* n. We will also say that $\mathsf{A}^n[\bullet] = \langle \langle \cdots \langle \bullet, \mathfrak{a}_{n-1} \rangle, \cdots, \mathfrak{a}_1 \rangle, \mathfrak{a}_0 \rangle$ are the levels of A lower than n, and we will denote by $\mathsf{A}^n[\mathsf{C}]$ the context obtained replacing C for the subcontext of A at width n (in our case replacing C for B).

We stress that in spite of their formalization as infinite lists, contexts are indeed finite objects. In fact, for any context A there is an index n such that the subcontext at width n is empty (i.e., $\mathsf{A} = \mathsf{A}^n[\varnothing]$). The *width* of a context is the least n for which $\mathsf{A} = \mathsf{A}^n[\varnothing]$.

Definition 3.4.2 (proper paths) A *proper path* in a sharing graph G is a path such that:

(i) every edge of the path is labeled with a context;

(ii) consecutive pairs of edges satisfy one of the following constraints:



(iii) the path does not contain bounce, i.e., none of the previous configurations can be traversed bottom-up immediately after it has been traversed top-down, and vice versa.

Given a proper path, the mapping between edges and contexts that the path induces is said a *proper context labeling* (proper labeling for short). Proper paths are equated up to contexts. That is, two proper paths having pairwise equal edges are considered equal, even when their contexts differ.

Remark 3.4.3 Proper paths never traverse a λ-node from the binding port to the context port (or *vice versa*). We will also see that proper paths are suitable representations of paths of the syntax tree of the λ-term denoted by the sharing graph. As a consequence the length of proper paths is always finite. This is the most manifest feature which distinguishes Lamping's paths from the paths we will study in Chapter 6.

Remark 3.4.4 (forward paths) Proper paths are not oriented. Nevertheless, in the following we shall mainly consider forward oriented proper paths, that is, oriented according to the natural top-down orientation of λ-term syntax trees. Thus, a *forward proper path* is a proper path whose λ and @ nodes are always traversed top-down, according to the drawing of Definition 3.4.2. Furthermore, for the rest of this chapter, and when not otherwise specified, paths will be forward oriented, that is, a part for contexts, they will verify all the composition rules of forward proper paths. More precisely, a *forward path* is a sequence of edges in which: (*i*) there are no bounces; (*ii*) @-nodes are traversed entering from the context port and leaving through the argument or to the function port; (*iii*) λ-nodes are traversed entering from the context port and leaving from the body port.

For instance, the arrows in the path of Figure 3.16 correspond to its forward orientation. It is indeed easy to check that the sequence of edges corresponding to a proper path is either forward oriented or the reverse of a forward path, say a backward path. Furthermore, any path starting at the root is forward oriented.

Exercise 3.4.5 Prove that any proper path has an infinite number of proper labelings.

Exercise 3.4.6 Let φ be a finite path. Prove that the predicate "ϕ has a proper labeling" is decidable.

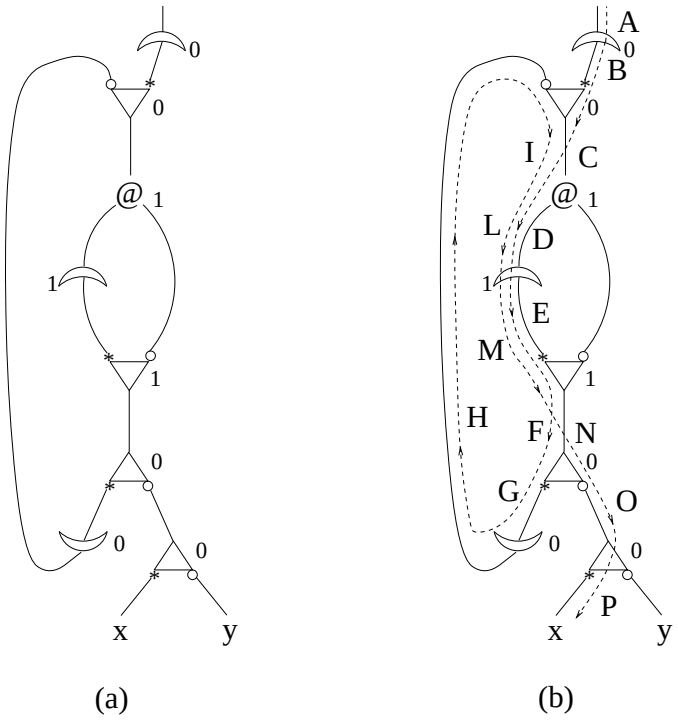


Fig. 3.16. A proper path.

Example 3.4.7 Figure 3.16(a) illustrates a (simplified) sharing graph segment which can occur near the end of the evaluation of

$$\lambda x.\, \lambda y.\, (\lambda z.\, (M\ N)\ P)$$

where

$$\begin{aligned} M &= \lambda u.\, (u\ (\lambda v.\, (u\ \lambda w.\, v))) \\ N &= \lambda i.\, (((z\ \lambda j.\, (i\ j))\ x)\ y) \\ P &= \lambda f.\, \lambda r.\, \lambda s.\, ((f\ r)\ (f\ s)) \end{aligned}$$

In Figure 3.16(b) we have drawn a proper path (the dashed line). The

labels of the edges are in order (being X an arbitrary context):

$$
\begin{array}{lcllcl}
\mathsf{A} & = & \mathsf{X} & \mathsf{H} & = & \langle \mathsf{X}, * \cdot \Box \rangle \\
\mathsf{B} & = & \langle \mathsf{X}, \Box \rangle & \mathsf{I} & = & \langle \mathsf{X}, \circ \cdot * \cdot \Box \rangle \\
\mathsf{C} & = & \langle \mathsf{X}, * \cdot \Box \rangle & \mathsf{L} & = & \langle \mathsf{X}, \circ \cdot * \cdot \Box \rangle \\
\mathsf{D} & = & \langle \mathsf{X}, * \cdot \Box \rangle & \mathsf{M} & = & \langle \langle \mathsf{X}, \Box \rangle, \circ \cdot * \cdot \Box \rangle \\
\mathsf{E} & = & \langle \langle \mathsf{X}, \Box \rangle, * \cdot \Box \rangle & \mathsf{N} & = & \langle \langle \mathsf{X}, * \cdot \Box \rangle, \circ \cdot * \cdot \Box \rangle \\
\mathsf{F} & = & \langle \langle \mathsf{X}, * \cdot \Box \rangle, * \cdot \Box \rangle & \mathsf{O} & = & \langle \langle \mathsf{X}, * \cdot \Box \rangle, * \cdot \Box \rangle \\
\mathsf{G} & = & \langle \langle \mathsf{X}, * \cdot \Box \rangle, \Box \rangle & \mathsf{P} & = & \langle \langle \mathsf{X}, * \cdot \Box \rangle, \Box \rangle
\end{array}
$$

We added the arrows along the path only for ease of reading, since proper paths are not oriented. Observe that the proper path of the example starts at the root of the sharing graph and terminates at the variable x, by traversing twice an application node from the context to the principal port. As a consequence, in the term represented by the graph in Figure 3.16(a), the variable x will be inside the first argument of two nested applications.

Exercise 3.4.8 Take the graph of Figure 3.16.

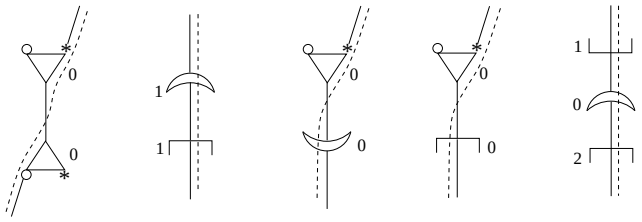
(i) Find a proper path starting at the root and terminating at y.

(ii) Is there a proper path starting at x and terminating at y?

(iii) Find all the proper paths starting at the root and terminating at a variable.

(iv) Show that any of the previous paths crosses twice the @-node.

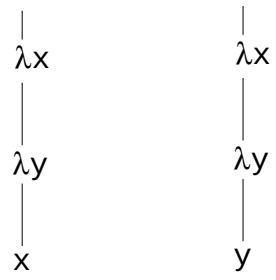
Example 3.4.9 Every path (the dashed lines) in the configuration below is not proper.



Notice that there are connections between principal ports of nodes which give rise to improper paths. Remark also that no rewriting rule has been defined for these connections.

3.5 Correctness of the algorithm

It is folklore that the standard syntax tree representation of terms may be uniquely described by the set of paths which start at the root and terminate at the nodes of the tree. These paths are called *access paths*. There is a subtlety for access paths in λ-calculus due to the presence of binders and bound variables. For example, the two terms $\mathsf{K} = \lambda x.\,\lambda y.\,x$ and $\mathsf{O} = \lambda x.\,\lambda y.\,y$ have the following syntax trees:



Notice that the two syntax trees have the same "structure" but differ for the node binding the leaf. Therefore, in the λ-calculus, two access paths are *equal* when they traverse nodes in the same order and the leaves are bound by corresponding binders. We will not be fussy on this issue, since we are sure the reader has kept the idea.

Remark 3.5.1 To simplify the presentation, in the following we will restrict without loss of generality to closed λ-terms (i.e., to terms in which all the variables are bound). Hence, any access path ending at a variable occurrence crosses a λ-occurrence binding that variable. As a consequence all the sharing graphs we will consider have just one entries, the *root* of the sharing graph, which we may assume to be a node with index $\emptyset$ (note that the use of the index $\emptyset$ corresponds to the fact that $[M] = [M]_\emptyset$).

Proper paths play in sharing graphs the same role that access paths play in the syntax tree representation. In fact, we are going to establish a one-to-one relation between proper paths starting at the root of a sharing graph and access paths of the associated term. There are two main problems in giving such a correspondence:

(i) *Paths created by* β*-reduction.* Every β-rule creates new access paths: the paths terminating at the variables bound by the abstraction are lengthened with the paths starting at the argument

of the application in the redex. A similar lengthening happens to proper paths when a β-interaction occurs. However, it is far to be obvious whether the two lengthenings preserve the correspondence between access paths and proper paths.

(ii) *Determining the binder of a variable occurrence.* As we said in the above discussion, in order to uniquely define an access path terminating at a bound variable, one has to determine the binder. This case is particularly hard in sharing graphs because there may be several occurrences along a proper path of the same λ-node. For example, the (simplified) graph segments in Figure 3.17 respectively occur near the end of the evaluation of $(\lambda x.(x\ \lambda y.(x\ (K\ y)))\ \underline{1})$ and $(\lambda x.(x\ \lambda y.(x\ (O\ y)))\ \underline{1})$, where $K = \lambda x.\lambda y.x$, $O = \lambda x.\lambda y.y$, and $\underline{1} = \lambda x.\lambda y.(xy)$. The dotted paths are proper paths terminating at a variable node and traversing twice the same λ. In Figure 3.17(a) the binder of the variable is the top λ, that is, the first one traversed by the path (the graph represents K); in Figure 3.17(b) the binder of the variable is the bottom λ, that is, the second one traversed by the path (the graph represents O).

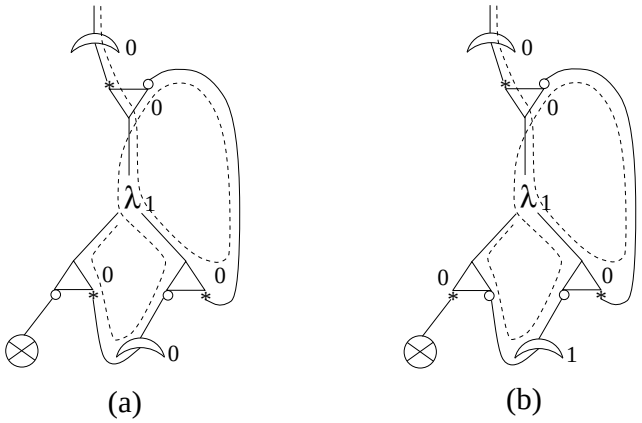


Fig. 3.17. Proper paths traversing twice the same binder

Exercise 3.5.2 Prove that (a part for a spurious bracket at the garbage node) the graphs of Figure 3.17 normalize to [K] and [O].

3.5.1 Properties of sharing graphs

The correctness of the algorithm relies on a few essential properties of proper paths and their context semantics. Let us start discussing them (their proof is postponed to section 3.5.6).

Remark 3.5.3 In this and in the next sections we shall refer to two distinct representations of λ-terms: sharing graphs and syntax trees. To avoid ambiguity we shall use λ-*node* and @-*node* to refer to nodes of the sharing graphs, λ-*occurrence* and @-*occurrence* to refer to nodes of the syntax trees. Furthermore, we shall use *proper node* to denote the root, or a λ or @ node, while we shall use *control node* to denote a bracket, croissant or fan.

The content of the first property, the so-called independence property, is rather technical. Besides, as we shall see farther, independence is the property ensuring the possibility to lengthen proper paths during β-reduction.

Proposition 3.5.4 (independence) *Given a proper path* φ *connecting two proper nodes whose indexes are respectively* $\mathfrak{m}$ *and* $\mathfrak{n}$*, let* A *and* B *be the initial and final contexts of* φ*.*

(i) *For any context* X*, the proper path* φ *has a proper context labeling assigning* $A^{\mathfrak{m}}[X]$ *to its initial edge.*

(ii) *According to the type of the port reached by* φ*, let us take* $\widehat{\mathfrak{n}} = \mathfrak{n}+1$ *when* φ *ends at the binding port of a* λ*-node, and* $\widehat{\mathfrak{n}} = \mathfrak{n}$ *otherwise (i.e.,* φ *ends at a context port). There is an index* $\widehat{\mathfrak{m}} \geq \mathfrak{m}$*, such that:*

  (a) *if* $B = B^{\widehat{\mathfrak{n}}}[C]$*, then* $A = A^{\widehat{\mathfrak{m}}}[C]$*;*

  (b) C *can be replaced by any context* Y*, i.e., for any context* Y*, there is a proper labeling of* φ *assigning* $A^{\widehat{\mathfrak{m}}}[Y]$ *and* $B^{\widehat{\mathfrak{n}}}[Y]$ *to the initial and final edges of* φ*;*

  (c) *moreover, when* φ *starts at the argument port of an* @*-node, we have indeed* $\widehat{\mathfrak{m}} > \mathfrak{m}$*.*

In other words, by the first item of independence, a forward path φ starting at an $\mathfrak{m}$-indexed proper node is proper (or not) independently from the initial context at width $\mathfrak{m}$. In particular, a path starting at the root is proper if and only if it has a proper labeling for any initial context. This is particularly relevant for the read-back algorithm. In fact, we will see that the read-back of a sharing graph (i.e., the λ-term

matching it) can be defined in terms of the proper paths starting at the root of the graph. By independence, looking for these proper paths we can start with any initial context and we do not need backtracking.

The second part of independence gives instead some details on the shape of contexts along a proper path.

As a matter of fact, finiteness of proper paths implies the existence of some part of the initial context of φ that we should find unmodified in the final context. Namely, for any index $\hat{n}$ big enough, we expect to have an index $\hat{m}$ for which the initial context of φ is $A = A^{\hat{m}}[C]$ and the final one is $B = B^{\hat{n}}[C]$. Independence clarifies how big we have to choose $\hat{n}$. In particular, whenever φ ends at a context port, no control node in φ has effect on the value of the final context at width n. While, in the case that φ ends at a binding port (i.e., it reaches a variable), some control nodes in φ might have written on level n of the final context. Indeed, in the latter case, level n contains the information relative to the control nodes belonging to a so-called loop of the λ-node λ reached by φ (see Definition 3.5.8).

Finally, let us note that what we find above $\hat{n}$ in the final context B was at width $\hat{m} \geq m$ in the initial context A. In some sense, the indexes of the proper nodes in a forward path are non-decreasingly ordered (compare with the indexes assigned by the translation of Figure 3.15). In fact, let us assume that φ end at a context edge, although the final index m might even be lower than n, the relevant information contained in the initial context (i.e., below level m) that has not been used along the path is recorded in the levels below n of the final context. Moreover, let us take any proper node traversed by the proper path φ, say an application @. This node splits φ in two proper subpaths $\varphi_1@\varphi_2$, to which we can separately apply independence. Then, let r be the index of @ and $D = D^r[E]$ be the context assigned to the edges of @. For the path φ_2, we get $B = B^{\hat{n}}[C]$ and $D = D^{\hat{r}}[C]$ for some $\hat{r} \geq r$, and thus $E = E^k[C]$ for some $k \geq 0$. At the same time, by independence of ϕ_1, we get $A = A^{\tilde{m}}[E]$, for some $\tilde{m} \geq m$. Summing up, $A = A^{\tilde{m}}[E^k[C]] = A^{\hat{m}}[C]$, with $\hat{m} = \tilde{m} + k \geq \tilde{m}$. In other words, interpreting $\hat{m}$ and $\tilde{m}$ as the indexes of the corresponding nodes reported to the initial node of φ (note that $\tilde{m}$ and $\hat{m}$ are definitely functions of r and n, respectively), we see that such reported indexes are non-decreasingly ordered along a proper path.

As a matter of fact, the previous consideration immediately forbids configurations like the ones in Figure 3.18. In all these cases the paths crossing the control nodes are proper; nevertheless, the indexes associ-

ated to them are incompatible with independence (in particular, the last remark on paths starting with an argument edge forbids the rightmost case).

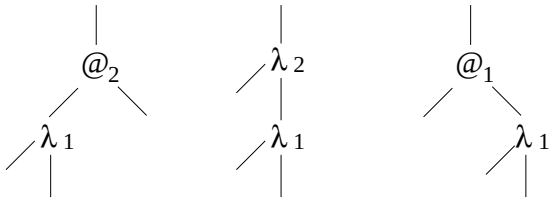


Fig. 3.18. Proper configurations incompatible with independence

The next two exercises formalize some of the previous considerations.

Exercise 3.5.5 Let φ be a forward proper path. Using independence property, prove that there is a proper labeling such that the width of the context at the context port of any $\mathfrak{n}$-indexed proper node is lower than $\mathfrak{n}$.

Exercise 3.5.6 Let φ be a forward proper path ending at an $\mathfrak{n}$-indexed proper node. Let A and B be respectively the contexts of the first and last edge of φ. Prove that:

(i) To any marker of a fan auxiliary port contained in B, say a $*$ symbol, we can associate a (unique) fan-in crossed by φ that wrote this $*$ in B, or conclude that such a $*$ was already present in A (we leave to the reader the exact formalization of the concepts "write in the context" and "was already present in the context").

(ii) Prove that any $*$ in B has not been written by a fan-in crossed by φ if and only it can be replaced by any other (possibly empty) sequence of $*$ and $\circ$ symbols (i.e., for any context B' obtained from B by the previous replacing, there is a proper context labeling assigning B' to the last edge of φ).

(iii) Using independence, prove that no fan-in crossed by φ can write a $*$ in the subcontext of B at width $\widehat{\mathfrak{n}}$ (i.e., any $*$ contained in a context X such that $\mathsf{B}^{\widehat{\mathfrak{n}}}[\mathsf{X}]$ was already present in the initial context A), where $\widehat{\mathfrak{n}}$ is defined as for independence.

(iv) Extend the previous results to brackets and croissants.

Another relevant consequence of independence is that, in certain cases, we can safely compose two proper paths φ_1 and φ_2 respectively ending

and starting at suitable n-indexed nodes, even when the final context of φ_1 and the initial context of φ_2 differ for the subcontext at width n. In particular, such a king of composition occurs when a β-reduction is performed. In this case any path φ_2 starting at the @-node in the redex is appended to one or more corresponding paths φ_1 ending at the binding port of the λ-node in the redex. What we have to prove is that this composition is sound; namely, that the meaningful part of the final context of φ_1 matches with the meaningful part of the initial context of φ_2 (i.e., the levels below n coincide), if and only if the paths p_1 and p_2, respectively corresponding to φ_1 and φ_2 in the matching syntax tree, are connected to the same redex (i.e., the corresponding reduction of the syntax tress creates an access path $p_1 \cdot p_2$ in the result). This is the purpose of the following nesting and transparency properties.

Proposition 3.5.7 (nesting) *Every proper path starting at the root node never reaches twice the context port of a n-indexed proper node with two contexts whose first n levels are equal.*

The first point to remark is that the nesting property gives a way for separating two occurrences of the same λ-node in a proper path. Let us consider for instance the two graphs in Figure 3.17. In both cases (a) and (b) the two occurrences of λ are accessed with two different contexts $\langle X, * \cdot \Box \rangle$ and $\langle X, * \cdot \Box \rangle$.

Secondly, the nesting property is necessary if we want to achieve that the length of any proper path starting at the root is finite, that at its turn is a necessary condition implied by the fact that we want to find a direct correspondence between proper paths and access paths.

Let us now come to the *crucial* notion of *binding loop*.

Definition 3.5.8 (loop) A *loop* is a proper path that starts at the body of a λ-node and terminates at the binding port of the same λ-node. A loop φ of an n-indexed λ-node is a *binding loop* if the initial context A and the final context B of φ are equal up to the subcontext at width n. Namely, φ is a binding loop when $A^n[\,.\,] = B^n[\,.\,]$.

As a consequence of the property of nesting, a loop has at most one suffix which is a binding loop. The underlying idea is that the occurrence of the λ-node binding a variable is uniquely determined by a binding loop. Because of the independence property, this fits well with the possible lengthening of the loop when the binder will be reduced (see Exercise 3.5.9). Note moreover that not every loop is a binding loop.

Let us consider again the example in Figure 3.17(a). As we already remarked, the first occurrence of λ is traversed with context $\langle \mathsf{X}, * \cdot \Box \rangle$, while the second occurrence is traversed with context $\langle \mathsf{X}, \circ \cdot \Box \rangle$. Now, when we come back to the λ-node, the context is $\langle \langle \mathsf{X}, \Box \rangle, * \cdot \Box \rangle$, that coincides with the context corresponding to the *first* traversal of λ, up to the subcontext at width 1. So, this is the real binder. Conversely, in case (b), the final context of the loop is $\langle \langle \mathsf{X}, \Box \rangle, \circ \cdot \Box \rangle$, that coincides with the context corresponding to the *second* traversal of λ, up to the subcontext at width 1.

Let us consider another example. The sharing graph in Figure 3.19 is a representation of λx.λy.(x y). A similar configuration may appear during the reduction of (λx.(x x) λx.λy.(x y)) (prove it as an exercise).

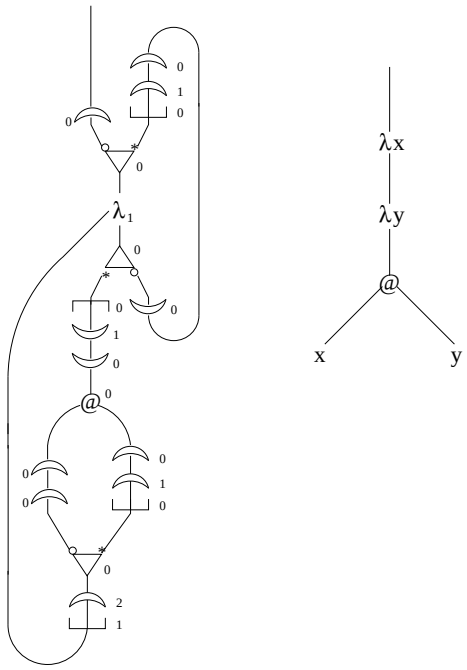


Fig. 3.19. Binding Loops.

Again, we have a double traversal of the λ-node before reaching the application: the first time. entering the λ-node with context $\langle \mathsf{X}, \circ \cdot \Box \rangle$; the second one entering it with context $\langle \mathsf{X}, * \cdot \langle \Box, \Box \rangle \rangle$. Then, when we eventually reach the application, we have two possibilities. If we follow the functional port of the application, we come back to the λ-node with

context $\langle\langle X, \langle\Box,\Box\rangle\rangle \circ \cdot \Box\rangle$. Thus, in this case, the variable is bound by the λ-occurrence corresponding to the first traversal of λ-node. On the other side, if we follow the argument port of the application, we come back to the λ-node with context $\langle X', \langle\Box, a_0\rangle\rangle, * \cdot \langle\Box,\Box\rangle\rangle$, where $X = \langle X', a_0\rangle$. In this case, the variable is thus bound by the λ-occurrence corresponding to the second traversal of the λ-node.

As already remarked, β-reduction corresponds to composition of proper paths; moreover, binding loops are the keystones of such a composition. The following exercise gives a first idea of that role of binding loops.

Exercise 3.5.9 Let $\varphi \cdot \nu_c \cdot w_a \cdot \psi$ and $\varphi \cdot \nu_c \cdot u \cdot \nu_b \cdot \phi \cdot w_\nu$ be two proper paths starting at the root and such that: (*i*) u is a β-redex; (*ii*) w_a is the argument edge of the @-node of u; (*iii*) ϕ is a binding loop. Let ν be the edge obtained merging the context edge ν_c and the body edge ν_b (i.e., ν is the contraction of the path $\nu_c \cdot u \cdot \nu_b$); and let w be the edge obtained merging the binding edge w_ν and the argument edge w_a (i.e., w is the contraction of the path $w_\nu \cdot u \cdot w_a$). Prove that the path $\varphi \cdot \nu \cdot \phi \cdot w \cdot \psi$ is proper.

So far, we have no way to ensure that any proper path ending at a variable corresponds to a properly bound variable. The next property of transparency guarantees that to reach a variable we also have to cross its binder.

Proposition 3.5.10 (transparency) *Every proper path starting at the root and terminating at a binding port has a suffix which is a binding loop.*

Let us now consider a more involved example. The sharing graph in Figure 3.20 is a representation of the λ-term:

$$\lambda y.(\,(\lambda w.((\lambda w.(y\ w))\ y)\ w)\ ((\lambda w.(y\ w))\ y)).$$

A similar configuration may appear during the reduction of the λ-term:

$$(\lambda x.\lambda y.\,(x\,(x\ y))\ \lambda z.((\lambda w.\,(z\,w))\ z)).$$

We have three possible traversals of the λ-node involved in the β-redex and several occurrences of variables bound by corresponding λ-occurrences. (For the sake of simplicity and referring to the syntax-tree on the right of Figure 3.20, let us enumerate these occurrences according

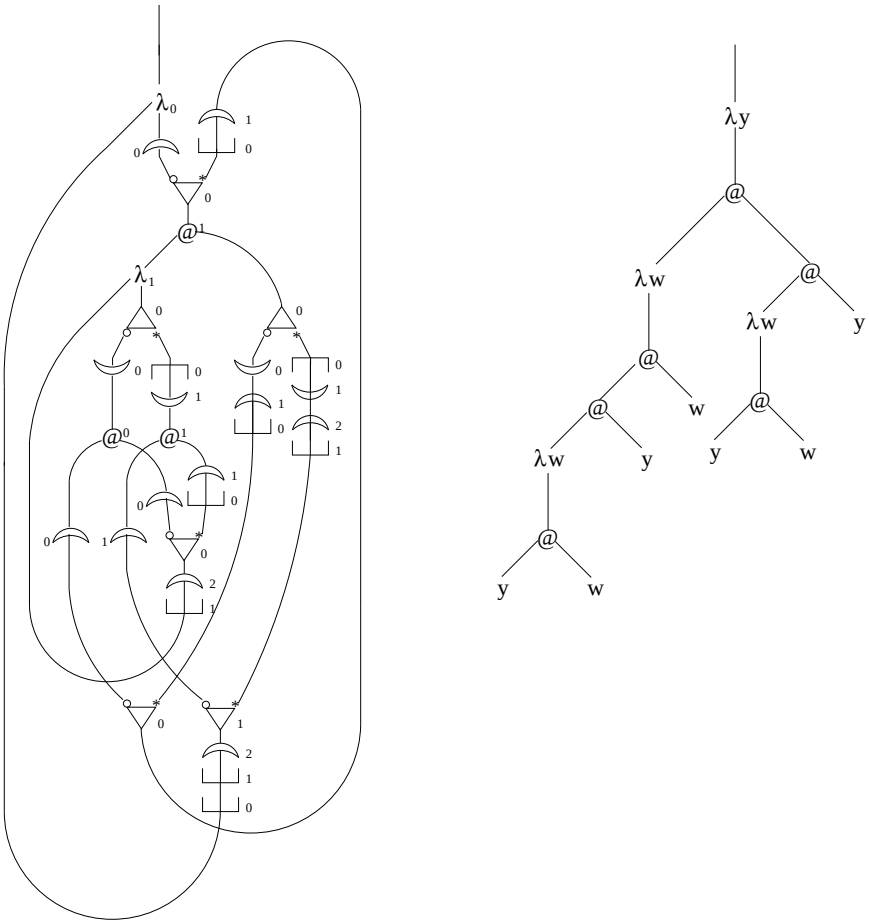


Fig. 3.20. An involved example.

to a depth-first-left-first traversal of the syntax tree.) Let $\langle\langle X, a_1\rangle, a_0\rangle$ be the initial context. The first occurrence of λw is reached with context

$$\langle\langle\langle X, a_1\rangle, a_0\rangle, \circ \cdot \Box\rangle,$$

the second with context

$$\langle\langle\langle X, a_1\rangle, a_0\rangle, * \cdot \langle\Box, \circ \cdot \Box\rangle\rangle$$

and the third with context

$$\langle\langle\langle X, a_1\rangle, a_0\rangle, * \cdot \langle\Box, * \cdot \Box\rangle\rangle.$$

The three (linear) occurrences of w are respectively reached with contexts:

$$\langle\langle\langle X, a_1\rangle, \langle\Box, a_0\rangle\rangle, \circ \cdot \Box\rangle$$
$$\langle\langle\langle X, a_1\rangle, \langle\Box, a_0\rangle\rangle, * \cdot \langle\Box, \circ \cdot \Box\rangle\rangle$$
$$\langle\langle\langle X, a_1\rangle, \langle\Box, a_0\rangle\rangle, * \cdot \langle\Box, * \cdot \Box\rangle\rangle$$

On the other side, we have a single occurrence of λy, reached with the initial context $\langle\langle X, a_1\rangle, a_0\rangle$ (therefore, *every* loop will be a binding loop). In this case, we have four loops, corresponding to the four occurrences of y. They are respectively reached with contexts:

$$\langle\langle X, a_1\rangle, a_0\rangle, \langle\langle\Box, \circ \cdot \Box\rangle, \circ \cdot \Box\rangle\rangle$$
$$\langle\langle X, a_1\rangle, \langle\langle\Box, * \cdot \langle\Box, a_0\rangle\rangle, \circ \cdot \Box\rangle\rangle$$
$$\langle\langle X, a_1\rangle, \langle\langle\Box, \circ \cdot \Box\rangle, * \cdot \langle\Box, a_0\rangle\rangle\rangle$$
$$\langle X, \langle\langle\Box, * \cdot \langle\Box, a_1\rangle\rangle, * \cdot \langle\Box, a_0\rangle\rangle\rangle$$

Suppose now to fire the redex in the sharing graph in Figure 3.20, that corresponds to three redexes in the associated λ-term. One may wonder what happens of the (distinguished!) arguments of these redexes. First of all, what are these arguments? They are defined by the paths which can be labeled with an initial context corresponding to the traversal of the respective application. Moreover, these contexts are just the three contexts corresponding to the three occurrences of λw defined above, and which, by independence, can be simplified into:

$$\langle X', \circ \cdot \Box\rangle$$
$$\langle X'', * \cdot \langle\Box, \circ \cdot \Box\rangle\rangle$$
$$\langle X''', * \cdot \langle\Box, * \cdot \Box\rangle\rangle$$

where X', X'' and X''' are arbitrary. But, by definition, no binding loop for λw can modify level 0 of that contexts. So, each "loop" will be connected to the right argument.

On the other hand, suppose to add an application with argument N (that will be at level 1) on top of our graph. In this case, the same argument N is the same for all binding loops for λy (here we have a single occurrence of λy with multiple occurrences of y). How do we distinguish among the different instances of N after the reduction? The key point is that two binding loops for the same traversal of an n-indexed λ-node are distinguished by their subcontexts at width n. The see this, let us compare two binding loops ϕ_1 and ϕ_2 for the same λ-node

starting from their binding port. Since the the two paths are distinct, they must eventually differ after a common postfix ζ. Moreover, it is indeed immediate to see that this must happen at a fan-in that ϕ_1 and ϕ_2 crosses through distinct auxiliary ports, say $*$ and $\circ$, respectively. According to what stated in Exercise 3.5.6, the fan-in preceding ζ writes a $*$ in the final context B_1 of ϕ_1, and analogously, it writes a $\circ$ in the corresponding position of the final context B_2 of ϕ_2. We can then conclude that B_1 differ from B_2, at least for the presence of a $*$ in the place of a $\circ$. By the way, when such a difference is in a level below n, the two binding loops corresponded to two distinct traversal of the λ-node, that is not the case in which we are interested now; otherwise, the two differences are in the subcontexts at width n. Moreover, by the independence property, the previous difference must necessarily be contained in the level n of B_1 and B_2, (i.e., if $B_1^n[\cdot] = B_2^n[\cdot]$ and $B_i = B_i^n[\langle C_i, b_i\rangle]$ with $i = 1,2$, then $b_1 \neq b_2$). For instance, the four binding loops of λy in the previous example differ at least for their level 0. To conclude this analysis, we invite the reader to give a formal proof a formal proof of the previous fact on binding loops exploiting the results in Exercise 3.5.6.

Exercise 3.5.11 Verify that the sharing graphs in Figure 3.17 and Figure 3.19 have the properties of independence, nesting, and transparency.

Exercise 3.5.12 The sharing graph in Figure 3.16 is part of the sharing graph in Figure 3.21. Prove that the properties of independence, nesting, and transparency hold for these sharing graphs.

3.5.2 Correspondence between sharing graphs and λ-terms

Definition 3.5.13 An access path p *corresponds to* a proper path φ (φ is supposed to start at the root) if p traverses proper nodes in the same order as φ and from the same ports. Moreover, if φ terminates at a binding port, the final variable of p is the variable bound by the unique abstraction along p corresponding to the λ-node of the binding loop.

A sharing graph G *matches with* a syntax tree N if there exists a bijective function f from proper paths of G from the root and to proper nodes and access paths in N such that:

(i) for every φ, $f(\varphi)$ corresponds to φ;

(ii) if $\varphi \cdot \psi$ and φ terminate at proper nodes, then there exists a path q such that $f(\varphi \cdot \psi) = f(\varphi) \cdot q$.

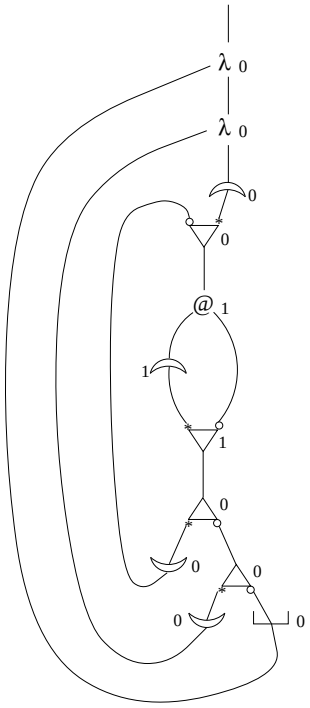


Fig. 3.21. Sharing graph for Exercise 3.5.12 and Exercise 3.5.14.

Exercise 3.5.14 Prove that the sharing graph of Figure 3.21 matches with the λ-term $\lambda y.\,\lambda x.\,((x\ x)\ (y\ y))$.

Since the function f between the proper paths of G and the access paths of the syntax tree N is a bijection, its inverse function f^{-1} maps any access path of N to a proper path starting at the root and ending at a proper node of G. Indeed, such a function f^{-1} is a function mapping any path of N into a proper path of G. In fact, let p be the access path of an edge e_p of N; taking $g(e_p) = f^{-1}(\xi)$, where φ and ξ are the unique proper paths such that $p = f(\varphi)$ and $p \cdot e_p = f(\varphi \cdot \xi)$; we can define $g(e_1 \cdots e_k) = g(e_1) \cdots g(e_k)$. It is not difficult to check that the restriction of g to the access paths of N coincide with f^{-1}.

We remark that, for any edge e, the proper path $g(e)$ contains only brackets and fans and connects two proper nodes. In particular, for any β-redex u of G there is a set $\mathcal{U} = \{u_1, \ldots, u_k\}$ of corresponding β-redexes of N such that $g(u_i) = u$, for $i = 1, \ldots, k$. We will see that the reduction

of $\mathfrak{u}$ corresponds to the simultaneous reduction of all the redexes in $\mathcal{U}$. But to prove this, we need to prove that all the rules of the algorithm are sound with respect to the correspondence between sharing graphs and syntax trees established by Definition 3.5.13.

Theorem 3.5.15 (correctness) *For any sharing graph reduction* $[M] \twoheadrightarrow G$, *there exists a corresponding* λ*-term reduction* $M \twoheadrightarrow N$ *such that* G *matches* N.

The tough part of Theorem 3.5.15 is the proof that β-rule preserves the right correspondence. In more details, let $[M] \twoheadrightarrow G' \xrightarrow{\mathfrak{u}} G$, where $\mathfrak{u}$ is a β-redex. Assuming that correctness holds for G', we have a λ-term N' that matches with G'. We need to find a λ-term N that matches with G and to prove that $N' \twoheadrightarrow N$. Let us analyze in more details why such a case is not direct.

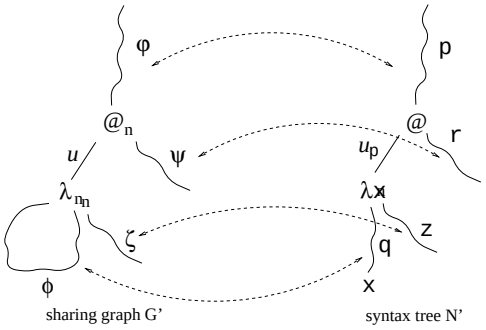


Fig. 3.22. Paths involved in a β-reduction.

The problem is illustrated by Figure 3.22 (in the following example the names of the paths will correspond to the ones used in such a figure). To start, let us assume that φ be the unique proper path ending at the @-node of the redex $\mathfrak{u}$. That is, the edge $\mathfrak{u}_p$ is the unique β-redex corresponding to $\mathfrak{u}$. As we already said, in this case we shall prove that $N' \xrightarrow{\mathfrak{u}_p} N$. Here, let us see how the paths of G' and N' are changed by the reduction of $\mathfrak{u}$ and $\mathfrak{u}_p$.

We can distinguish two cases: (*i*) the path ends at a proper node in the body of the redex; (*ii*) the path ends at a proper node in the argument of the redex. In fact, the reduction leaves unchanged all the paths with a different shape.

In the first case, the proper path crosses the edge $\mathtt{u}$ and the corresponding access path crosses the edge $\mathtt{u}_p$. The reduction erases the edge $\mathtt{u}$ of the proper path $\varphi \cdot \mathtt{u} \cdot \zeta$ and the edge $\mathtt{u}_p$ of the corresponding access path $\mathtt{p} \cdot \mathtt{u}_p \cdot \mathtt{z}$, giving $\varphi \cdot \zeta$ and $\mathtt{p} \cdot \mathtt{z}$. Thus, we see that properness of $\varphi \cdot \zeta$ follows from properness of $\varphi \cdot \mathtt{u} \cdot \zeta$ and that the correspondence between the paths is preserved, i.e., $\varphi \cdot \zeta$ and $\mathtt{p} \cdot \mathtt{z}$ correspond. (Note the small abuse of notation in the last writing. In fact, after the reduction, the last edge of φ and the first of ζ are merged, and similarly for the last edge of $\mathtt{p}$ and $\mathtt{z}$.)

In the second case, the difficult one, the proper path crosses the @-node of the redex from the context port to the argument port, and similarly for the corresponding access path and @-occurrence. We have then a proper path $\varphi \cdot \psi$ and a corresponding access path $\mathtt{p} \cdot \mathtt{r}$. Here, rather than erasing edges, the β-reduction of $\mathtt{u}$ inserts a path in the place of the @-node and a path in the place of the @-occurrence. In fact, for any binding loop ϕ, and any corresponding path $\mathtt{q}$ (connecting the matching λ-occurrence to an occurrence of its bound variable), we get a new proper path $\varphi \cdot \phi \cdot \psi$, and a new access path $\mathtt{p} \cdot \mathtt{q} \cdot \mathtt{r}$. The situation is thus no more direct as in the first case. But, by definition of binding loop and independence, the insertion of the loop ϕ has no impact on properness (see Exercise 3.5.9).

Unfortunately, the reasoning is no more so straight when the restriction on $\mathtt{u}$ is removed. In the general case, the paths φ, ζ, ψ and ϕ may cross at their turn the @-node of the redex, either towards the body of the redex than towards the argument. In particular, the second case of the simplified example becomes even more involved, for the loop ϕ is no more a proper path of G but a path that might have been expanded at its turn. To take into account this sort of recursive expansion, we need to extend the definition of proper path.

3.5.3 Expanded proper paths

An *expanded forward path* is a sequence of edges concatenated according to the construction rules of forward paths plus the following composition rule: the sequence $\varphi \cdot \underline{\mathtt{u}} \cdot \psi$ is an expanded forward path when $\mathtt{u}$ is a β-redex, φ ends at the binding port of the λ-node of $\mathtt{u}$, and ψ starts at the argument port of the @-node of $\mathtt{u}$. Let us note that in the previous composition rule, the occurrence $\underline{\mathtt{u}}$ of the β-redex $\mathtt{u}$ is crossed following the backward orientation. (The underlining has been introduced to emphasize this point.) Let us also note that such underlined β-redex

are the only edges that an expanded proper path crosses according to backward orientation.

The idea is that *expanded forward path* are just the paths that can be contracted into forward paths by the firing of some redex. We shall prove that, provided independence, nesting, and transparency for forward proper paths, they hold for expanded paths as well. This is essentially enough to prove Theorem 3.5.15.

Definition 3.5.16 The set of the (forward) *expanded proper paths* (starting at the root) is the smallest set of expanded forward paths such that:

(i) Any proper path starting at the root is an expanded proper path.

(ii) $\varphi \cdot \mathfrak{u} \cdot \phi \cdot \underline{\mathfrak{u}} \cdot \psi$ is an expanded proper path when:

  (a) $\mathfrak{u}$ is a β-redex;

  (b) $\varphi \cdot \psi$ is an expanded proper path and ψ starts at the argument port of the @-node of $\mathfrak{u}$;

  (c) ϕ is a binding loop and $\varphi \cdot \mathfrak{u} \cdot \phi$ is an expanded proper path.

The notion of proper labeling given for proper paths applies to expanded paths too, assuming that, in addition to the constraints of Definition 3.4.2, any context edge of a λ-node following a binding edge is labeled by the same context of the binding edge. Also the definition of binding loop can then be extended to expanded paths. Namely, an *expanded binding loop* is an expanded path connecting the body port to the binding port of the same λ-node with a proper labeling such that the initial and final contexts are equal up to the subcontext at width $\mathfrak{n}$, where $\mathfrak{n}$ is the index of the loop λ-node.

We stress that an expanded proper path may contain two occurrences of the same edge $\mathfrak{u}$ with the same context. In fact, for any loop ϕ, the contexts A and B of the two occurrences of $\mathfrak{u}$ in the expanded path $\varphi \cdot \mathfrak{u} \cdot \phi \cdot \underline{\mathfrak{u}} \cdot \psi$ may be equal. Moreover, even when $A \neq B$, the contexts A and B are definitely equal up to the subcontext at width $\mathfrak{n}$, where $\mathfrak{n}$ is the index of the λ-node of $\mathfrak{u}$. Nevertheless, this does not contradict the nesting property, as nesting property compares the contexts of forward oriented edges only.

The analogous of the properties of independence, nesting, and transparency hold for expanded proper paths assuming that such properties hold for proper paths, as claimed by the next lemma.

Lemma 3.5.17 *The properties of forward proper paths given in Propo-*

sition 3.5.4, Proposition 3.5.7 and Proposition 3.5.10 imply the following properties of expanded proper paths:

(independence) *Given a proper context labeling of an expanded proper path φ ending at an $\mathfrak{n}$-indexed proper node, let B be the final context of φ and A be the context of an edge ν in φ following an $\mathfrak{m}$-indexed proper node.*

(i) *For any context X, the proper path φ has a proper context labeling assigning $\mathsf{A}^{\mathfrak{m}}[\mathsf{X}]$ to the edge ν.*

(ii) *According to the type of the port reached by φ, let us take $\widehat{\mathfrak{n}} = \mathfrak{n}+1$ when the last edge of φ is a reversed β-redex or φ ends at a binding port, and $\widehat{\mathfrak{n}} = \mathfrak{n}$ otherwise (i.e., φ ends at a context port). There is an index $\widehat{\mathfrak{m}} \geq \mathfrak{m}$, such that:*

(a) *if $\mathsf{A} = \mathsf{A}^{\widehat{\mathfrak{m}}}[\mathsf{C}]$, then $\mathsf{B} = \mathsf{B}^{\widehat{\mathfrak{n}}}[\mathsf{C}]$;*

(b) *C can be replaced by any context Y, i.e., for any context Y, there is a proper labeling of φ assigning $\mathsf{A}^{\widehat{\mathfrak{m}}}[\mathsf{Y}]$ to ν and $\mathsf{B}^{\widehat{\mathfrak{n}}}[\mathsf{Y}]$ to the final edge of ψ;*

(c) *moreover, when ν is a reverse redex or the argument edge of an @-node an argument port, we have indeed $\widehat{\mathfrak{m}} > \mathfrak{m}$.*

(nesting) *An expanded proper path never reaches twice the context port of an $\mathfrak{n}$-indexed proper node with two contexts whose first $\mathfrak{n}$ levels are equal.*

(transparency) *Every expanded proper path terminating at a binding port has a suffix which is an expanded binding loop.*

The only difference with respect to to forward proper paths is the statement of independence. In fact, for expanded proper paths always start at the root, we had to rearrange the formulation of independence. Indeed, in the case of non-expanded proper paths the formulations of independence in Lemma 3.5.17 and Proposition 3.5.4 are indeed equivalent: it suffices to note that any subpath of a (non-expanded) proper path is proper, and to note that independence is trivially compositional for (non-expanded) proper paths.

Let us also remark that also the formulation of independence in Lemma 3.5.17 implies the existence of a proper labeling for any context assigned to the initial edge of an expanded proper path (remind that an expanded proper path always starts at the root).

The proof of Lemma 3.5.17, will be given in section 3.5.5. Besides,

Lemma 3.5.17 allows to finally prove that sharing graph reduction is correct.

Exercise 3.5.18 Prove that any prefix of an expanded proper path is an expanded proper path.

Exercise 3.5.19 Let ξ be an expanded proper path such that $\phi = \varphi \cdot \mathfrak{u} \cdot \phi \cdot \underline{\mathfrak{u}} \cdot \psi$, for some expanded binding loop ϕ and some β-redex $\mathfrak{u}$.

(i) Prove that $\varphi \cdot \mathfrak{u} \cdot \psi$ and $\varphi \cdot \phi$ are expanded proper paths.

(ii) Let ν and ϕ' be another β-redex and another expanded binding loop such that $\xi = \varphi' \cdot \nu \cdot \phi' \cdot \underline{\nu} \cdot \psi'$. Assume that Lemma 3.5.17 holds for the expanded proper paths $\varphi \cdot \mathfrak{u} \cdot \psi$, $\varphi \cdot \phi$, $\varphi' \cdot \nu \cdot \psi'$, and $\varphi' \cdot \phi'$. Prove that $\phi \cap \phi' \neq \emptyset$ implies either $\phi \subseteq \phi'$ or $\phi' \subseteq \phi$.

Exercise 3.5.20 An alternative definition of expanded binding loop might be given by a constructive technique similar to the one used to define expanded proper paths. Namely, we might say that an expanded path ϕ starting at the body port of a λ-node is an expanded binding loop when:

(i) ϕ is a binding loop, or

(ii) $\phi = \varphi \cdot \mathfrak{u} \cdot \theta \cdot \underline{\mathfrak{u}} \cdot \psi$, where

  (a) $\mathfrak{u}$ is a β-redex;

  (b) $\varphi \cdot \psi$ is an expanded binding loop and ψ starts at the argument port of the @-node of $\mathfrak{u}$;

  (c) θ is an expanded binding loop and $\varphi \cdot \mathfrak{u} \cdot \theta$ is an expanded proper path.

Prove that the latter definition of expanded binding loop is equivalent to the original one. Furthermore, prove that expanded binding loops might be used to reformulate the definition of expanded proper paths, replacing "ϕ is an expanded binding loop" for "ϕ is a binding loop".

3.5.4 Proof of Theorem 3.5.15 (correctness)

Before to enter into the details of the proof, let us introduce some terminology.

First of all, the definition of expanded path can be reformulated for access paths too, simply replacing "access" for any occurrence of "proper" Definition 3.5.16. We get in this way that:

(i) Any access path is also an *expanded access path*.

(ii) Any sequence $\mathtt{p}\cdot\mathtt{u}\cdot\mathtt{q}\cdot\underline{\mathtt{u}}\cdot\mathtt{r}$ is an expanded access path when: (*i*) $\mathtt{u}$ is a β-redex; (*ii*) $\mathtt{p}\cdot\mathtt{u}\cdot\mathtt{q}$ and $\mathtt{p}\cdot\mathtt{r}$ are expanded access paths; (*iii*) $\mathtt{q}$ is an expanded access loop (i.e., it starts at the body port of a λ-node and ends at a variable occurrence bound by the previous λ-node).

Secondly, given a syntax tree β-reduction $\mathsf{N}' \to \mathsf{N}$ let us define a residual relation between edges and paths. Namely, an edge $\mathtt{u}$ of N is a *residual* of an edge $\mathtt{e}'$ of N' when $\mathtt{u}$ is a copy of the edge $\mathtt{u}'$. It is readily seen that the only edges of N without a copy in N' are: (*i*) the context-body edge connecting the context node of the redex (the one above the @-occurrence) to the body node (the one below the λ-occurrence); (*ii*) the binding-argument edges connecting an occurrence of the variable of the redex to the argument node (the one at the argument port of the @-occurrence). Besides, any new context-body edge can be seen as the merging of a context edge and of a body edge, while any binding-argument edge can be seen as the merging of a binding edge (one of the edges connected to the binding port of the redex) and of an argument edge. According to this remark, let $\mathtt{u}_{\mathtt{cb}}$ be a context-body edge originated reducing a β-redex edge $\mathtt{u}$ whose corresponding context and body edges are respectively $\mathtt{u}_{\mathtt{c}}$ and $\mathtt{u}_{\mathtt{b}}$. The relation $\mapsto$ can be extended taking $\mathtt{u}_{\mathtt{c}}\cdot\mathtt{u}\cdot\mathtt{u}_{\mathtt{b}} \mapsto \mathtt{u}_{\mathtt{cb}}$. Similarly, let $\mathtt{u}_{\mathtt{va}}$ be a new binding-argument edge. Also in this case, we can extend the relation $\mapsto$ taking $\mathtt{u}_{\mathtt{v}}\cdot\mathtt{u}\cdot\mathtt{u}_{\mathtt{a}} \mapsto \mathtt{u}_{\mathtt{va}}$, where $\mathtt{u}_{\mathtt{v}}$ is the binding edge and $\mathtt{u}_{\mathtt{a}}$ is the argument edge. In both the previous cases, we have associated a path to an edge of the result of the reduction; in particular, let us note that the ancestor of a binding-argument edge is an expanded path. With this extension the inverse of the relation $\mapsto$ becomes a function $\hookleftarrow$ from the edges of N to paths of N'. Furthermore, when $\mathsf{N}' \twoheadrightarrow \mathsf{N}$, the previous relations induce a relation $\overset{*}{\mapsto}$ and a corresponding map $\overset{*}{\hookleftarrow}$ that are just the transitive closure of $\mapsto$ and $\hookleftarrow$, respectively; these maps extend in the natural way to expanded access paths as well. The relevant point of the last extension to expanded paths is that there is a straight correspondence between the access paths of the result of a reduction and the expanded access paths of the initial syntax tree, as proved by the result in the next exercise.

Exercise 3.5.21 Let $\rho : \mathsf{N}' \twoheadrightarrow \mathsf{N}$ be a reduction relative to a set of redexes of N' (i.e., for any redex $\mathtt{u}$ reduced by ρ, we have that $\mathtt{u} \overset{*}{\hookleftarrow} \mathtt{u}'$, for some redex $\mathtt{u}'$ of N'). Prove that for any access path $\mathtt{p}$ of N there is a unique expanded access path $\mathtt{p}'$ of N' such that $\mathtt{p} \overset{*}{\hookleftarrow} \mathtt{p}'$.

The previous definitions of residual relation ($\mapsto$) and ancestor map ($\hookleftarrow$) apply immediately also to sharing graphs. Furthermore, since in this case a β-reduction creates just one new context-body edge and one new binding-argument edge, we see that the restriction of $\mapsto$ to the maximal paths containing control nodes only gives a function, or equivalently, the corresponding restriction of $\hookleftarrow$ is injective.

Let us now go on with the proof, proceeding by induction on the length of the derivation $[\mathsf{M}] \twoheadrightarrow \mathsf{G}$.

3.5.4.1 Base case

The matching is immediate: every path obtained starting at the root and traversing $\mathsf{G} = [\mathsf{M}]$ following the natural orientation of G (i.e., going top-down with respect to the translation given in Figure 3.15) is proper. Furthermore, any of such paths is isomorphic to an access path of the syntax tree of M.

3.5.4.2 Induction case

Let $[\mathsf{M}] \twoheadrightarrow \mathsf{G}' \overset{\mathfrak{u}}{\to} \mathsf{G}$. If $\mathfrak{u}$ is a rule involving at least a control node, then $\mathfrak{u}$ does not affect the expression represented by the graph. Indeed, the set of expanded proper paths connecting proper nodes does not changes (the reader is invited to check this statement). Therefore, correctness follows by induction hypothesis. Thus, the crucial case is β-reduction.

Let us assume that $\mathfrak{u}$ is a β-redex. If N' is the term represented by G' and f' is the one-to-one correspondence between them, we have to find a term N, with $\mathsf{N}' \twoheadrightarrow \mathsf{N}$, such that G and N match through a one-to-one correspondence f. By induction hypothesis we assume to know the functions g' and f', and the λ-term N' corresponding to G'. We need to find g, f and N.

(i) To define N, let $\mathcal{U} = \{\mathfrak{u}_1, \cdots, \mathfrak{u}_h\}$ be the β-redexes of N' corresponding to $\mathfrak{u}$. The λ-term N is the result of any development of $\mathcal{U}$ (for a definition of development see section 5.1.1).

(ii) To define g we can proceed as follows: for any edge e of N, let us take the path p of N' such that $\mathsf{e} \overset{*}{\hookleftarrow} \mathsf{p}$; then, let us take the image $\mathsf{g}'(\mathsf{p})$ of such a path in G'; we can easily see that there is a unique path φ of G such that $\varphi' \mapsto \varphi$; we define $\mathsf{g}(\mathsf{e}) = \varphi$.

It is readily seen that this procedure associates to each access path p of N an expanded proper path φ' starting at the root of G' and then a path φ starting at the root of G. Furthermore, let us note that φ is obtained from φ' merging into a unique edge sequences $\mathfrak{u}' \cdot \mathfrak{u} \cdot \mathfrak{u}''$ whose

contexts are always equal. We can thus conclude, by the proper labeling property of expanded proper paths (Lemma 3.5.17), that φ is a proper path.

We now have to prove that any proper path of G is image of an access path of N. The first point is to show that any proper path φ of G is image of exactly one expanded proper path of G'. To prove this we can proceed by induction on the number of binding-argument edges contained in any proper path φ of G. The base case is direct. In fact, when φ does not contain binding-argument edges, it is readily seen that there is a proper (non-expanded) path φ' of G such that $\varphi \leftarrowtail \varphi'$. Hence, let us assume that $\varphi = \psi \cdot e_{va} \cdot \zeta$, where e_{va} is the binding-argument edge introduced by the reduction of u, and $e_{va} \leftarrowtail e_v \cdot u \cdot e_a$. Let $\psi' \cdot e_v$ be the expanded path of G' such that $\psi' \leftarrowtail \psi$. By the induction hypothesis (the number of binding-argument edges of ψ is less the number of binding-argument edges of φ), $\psi' \cdot e_v$ is an expanded proper path from the root to the binding port of u. Hence, by the nesting property, there is a suffix ϕ' of ψ' (i.e., $\psi' = \eta' \cdot \phi'$) such that $\phi' \cdot e_v$ is an expanded binding loop connecting the body port of u to its binding port. Furthermore, let us also assume that the occurrence of e_{va} under analysis is the last one contained in φ. Under this additional assumption, the forward path ζ' such that $\zeta \leftarrowtail \zeta'$ is proper (note that ζ is not expanded) and, by the independence property, we also see that $\eta' \cdot e_c \cdot e_a \cdot \zeta'$ is an expanded proper path. We conclude then that $\varphi' = \eta' \cdot e_c \cdot \phi' \cdot e_a \cdot \zeta'$ is an expanded proper path (see Exercise 3.5.20).

The next point is to show that any expanded proper path of G' determines a unique access path of N. To do this, let us note that, since φ is a (non-expanded) proper path, the expanded binding loops used in the construction of φ' are relative to the λ-node of u. As a consequence, the residual p of the expanded access path p' of N' such that $g(p') = \varphi'$ is an access path of N. (The detailed proof of the last assertion is left to the reader. To this purpose we also suggest to see Exercise 3.5.21). Hence, by construction, we have that $f(\varphi) = p$.

Exercise 3.5.22

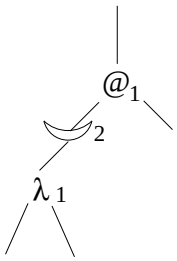
(i) Define a procedure which takes a set of access paths and gives the syntax tree of a λ-expression.

(ii) Define a procedure (the read-back) which takes a sharing graph G and gives the syntax tree of the term associated to G. (*Hint:* Define a procedure which gives the set S of proper paths of G which start at the root and terminate at proper nodes. Then,

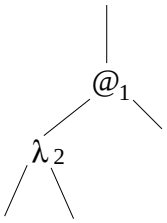
define a procedure which takes S and gives the corresponding set of access paths. Finally, use the procedure in the previous item.)

Remark 3.5.23 Theorem 3.5.15 might be stated in a stronger form, in order to ensure that when $[M] \twoheadrightarrow G$ and G is in normal form (there is no interaction), the syntax tree matching with G is in normal form too. To this aim one has to prove that the following two graph segments never appear:

(i) An $\mathfrak{m}$-indexed control node in front of the principal port of an $\mathfrak{n}$-indexed proper node and $\mathfrak{m} \geq \mathfrak{n}$. For instance:



(ii) Two proper nodes of different index which interact, as in:



Remark that the two paths connecting the nodes @ and λ in the above pictures are both proper.

Exercise 3.5.24 Referring to the previous remark:

(i) Prove that graph segments as in the first item never appear. (*Hint:* Use the property of independence.)

(ii) (*Difficult*) Prove that graph segments as in the second item never appear. (*Hint:* Define a function ℓ which takes a proper path traversing control nodes only and gives the "level change" performed by the path. That is, the traversal of a croissant from the auxiliary port to the principal port increases the level by 1; the traversal of a bracket from the auxiliary port to the principal port decreases the level by 1, etc. Hence, prove that if φ connects

the function port of an n-indexed @-node and the context of an m-indexed λ-node, then $n + \ell(\varphi) = m$.)

3.5.5 Proof of Lemma 3.5.17 (properties of expanded proper paths)

All the proofs but the one of nesting are by induction on the definition of expanded proper path.

The base case holds by hypothesis. So, let $\varphi = \zeta \cdot u \cdot \phi \cdot \underline{u} \cdot \psi$ be a decomposition of the expanded proper path φ. That is, u is a β-redex, $\zeta \cdot u \cdot \phi$ and $\zeta \cdot \psi$ are expanded proper paths, and ϕ is a binding loop.

The properties are proved in order.

3.5.5.1 Independence

First of all let us see how to build a proper context labeling of φ for any initial context. By induction hypothesis independence holds for both $\zeta \cdot u \cdot \phi$ and $\zeta \cdot \psi$; in particular, this also means that they have a proper labeling for any initial context. Let us choose the same initial context for the two paths. It is immediate that in both the paths the contexts of the edges in ζ coincide. By definition of binding loop, if D is the final context of ζ and r is the index of the nodes in the redex u, the final context of ϕ in the previous path is equal to $D^r[E]$, for some context E. By induction hypothesis, there is a proper context labeling for $\zeta \cdot \psi$ in which the context at width r in D is replaced by an arbitrary context; in particular, it means that there is labeling yielding $G = D^r[E]$ to the initial edge of ψ. We can then use this labeling to append ψ to $\zeta \cdot u \cdot \phi$ through $\underline{u}$.

The previous construction proves at the same time the existence of a proper assignment for any X replacing the context at width $\widehat{m}$ of ν, when ν is in $\zeta \cdot u \cdot \phi$ or in ψ (apply induction hypothesis to the subpath in which ν is and take the corresponding initial context for ζ). For the case in which $\nu = \underline{u}$ instead, it suffices to note that the context of $\underline{u}$ is always the same of the first edge of ψ, which starts at the argument edge of the @ node in the redex (i.e., $r = n$ and $\widehat{n} = r + 1$).

In the case in which ν is in ψ, the rest of the proof follows immediately by induction hypothesis. Thus, let us assume that ν is in $\zeta \cdot u \cdot \phi$.

Let $E = \langle F, a \rangle$. By the initial construction and by induction hypothesis, we have that $A = A^{\widetilde{m}}[F]$, for some $\widetilde{m} \geq m$; and at the same time, taking $B = B^{\widehat{n}}[C]$, there is $k \geq 0$ for which $F = F^k[C]$ (i.e., since the fist edge of ψ is an argument edge, $G = D^r[\langle F^k[C], a \rangle] = G^{\widehat{r}}[C]$, for some

$\hat{r} = r + k > r$). Summing up, the last context of φ is $B^{\hat{n}}[C]$ while the context of ν is $A = A^{\tilde{m}}[F] = A^{\tilde{m}}[F^k[C]] = A^{\hat{m}}[C]$, with $\hat{m} = \tilde{m} + k \geq m$. The other properties follows then by inspection of the latter construction. In particular, replacing Y for C, we have $D^r[\langle F^k[Y], a\rangle]$ at the beginning of ψ; but, by induction hypothesis, $F = F^k[C]$ can be simultaneously replaced by any context in both $G = D^r[\langle F, a\rangle]$ and $A = A^{\tilde{m}}[F]$; thus, putting $F^k[Y]$ for F, we conclude.

3.5.5.2 Nesting

The proof is by contradiction. Without loss of generality, we can take $\varphi = \eta \cdot u \cdot \chi \cdot u$, where u is the context edge u of some proper node and χ is not empty (i.e., the two occurrences of u are distinct); we can assume that the context assigned to the two occurrences of u is the same and, moreover, that φ is the shortest path with this property. Let μ' and μ'' be respectively the postfixes of η and χ composed of control nodes only. Since φ is the shortest path for which nesting does not hold, we see that μ' and μ'' must differ; moreover, the only possibility is that after a common postfix these paths contain a fan-in that μ' crosses entering through the $\circ$-port and μ'' passing through the $*$-port. Summarizing, we have two proper paths $\mu' \cdot u = \xi' \cdot \xi \cdot u$ and $\mu'' \cdot u = \xi'' \cdot \xi \cdot u$ such that:

(i) the proper paths $\xi' \cdot \xi \cdot u$ and $\xi'' \cdot \xi \cdot u$ cross control nodes only and end at the context port of an n-indexed proper node;
(ii) ξ' and ξ'' start at the ports of some proper nodes (note that they might also be the same port of the same proper node);
(iii) ξ' and ξ'' end at distinct auxiliary ports of the same fan-in, say that they reach the $\circ$ and the $*$-port, respectively.

Let A_1, B_1 and A_2, B_2 be the initial and final contexts of the proper paths $\mu' \cdot u \cdot u$ and $\mu'' \cdot u$, respectively. By contradiction of nesting, there is some context assignment such that $B_1^n[\,\cdot\,] = B_2^n[\,\cdot\,]$. At the same time, the shape of μ' and μ'' implies that $B_1 \neq B_2$ for any context assignment. But, see Exercise 3.5.6, this contradict independence (note that to prove independence we did not use nesting).

3.5.5.3 Transparency

By induction hypothesis, there is an expanded binding loop θ such that $\zeta \cdot \psi = \xi \cdot w \cdot \theta$. We can then see that this induces a decomposition of φ such that $\varphi = \xi' \cdot w \cdot \theta'$, where ϕ belong to ξ' or to θ' according to the position of w with respect to ζ.

3.5.6 Proof of the sharing graph properties

By induction on the length of the derivation ending with the sharing graph G.

3.5.6.1 Induction case: $[M] \twoheadrightarrow G' \overset{u}{\to} G$

Let u be a rule involving a control node. We invite the reader to check case-by-case that these rules do not invalidate the properties.

When u is a β-reduction, we have already seen in the proof of Theorem 3.5.15 that each proper path φ of G starting at the root is the residual of an expanded proper path φ' of G'. Furthermore, since φ is obtained from φ' replacing any triple $u' \cdot u \cdot u''$ with a single edge. It is readily seen that φ has the independence, nesting and transparency properties if and only if φ' has such properties. We conclude thus by induction hypothesis.

3.5.6.2 Base case: $G = [M]$

The control nodes of G are all in the paths connecting the binding ports of λ-nodes to the nearest proper node. Because of this it is immediate to see that all the proper path φ ending at the context port of a proper node contains proper nodes only. Hence, all the contexts of φ are equal. Furthermore, by analysis of the translation of Figure 3.15, we see that the levels of any node of φ is greater or equal than the level m of the initial node, and that when φ starts from an argument port they are indeed greater than m. These observations suffice to prove independence.

Nesting property is void. In fact, in G we cannot have a proper path reaching twice the context port of the same proper node.

The proof of transparency is by induction on the definition of the translation $[\,.\,]$. We leave it to the reader, see Exercise 3.5.25.

Exercise 3.5.25 Let $\underline{\mu}$ be the backward path connecting a binding port to a croissant of $[M]$ (note that μ crosses control nodes only). The offset p of μ (being μ the forward path obtained reversing $\underline{\mu}$) is the number of square brackets contained in μ. Let $\phi = \psi \cdot \mu$ be a loop of $[M]$. Prove that ϕ is a binding loop and that if A and B are respectively the initial and final contexts of ϕ, then

$$B = \begin{cases} A^n[\langle C, \langle \cdots \langle c_p \cdot \Box,\, c_{p-1} \cdot a_{p-1} \rangle, \cdots,\, c_0 \cdot a_0 \rangle\rangle] & \text{when } p > 0 \\ A^n[\langle C, \Box \rangle] & \text{when } p = 0 \end{cases}$$

where: (i) $a_i = A(n+i)$ and (ii) c_i is a (possibly empty) string of $\circ$ and $\star$, for $i = 0, \ldots, p-1$; (iii) $A = A^{n+p}[C]$.

4

Optimal Reductions and Linear Logic

There exists a deep relation between Lamping's optimal graph reduction technique and Girard's Linear Logic. Although this relation becomes really intriguing only considering the notion of path, and the so-called geometry of interaction for Linear Logic, we anticipate in this chapter the main ideas behind this correspondence, for we believe it could provide some more operational intuition on Lamping's Algorithm. Besides, the existence of a connection between Lamping's technique and Linear Logic is not as surprising as it might appear at a first glance. In fact, Lamping's algorithm provides a sophisticated way for handling sharing in lambda terms. This requires a deep investigation of the operation of duplication (or contraction, in logical terms), and a clean approach to the notion of sharable entity, which are also the main topics of Linear Logic: duplication (contraction) is generally avoided unless on special "sharable" objects equipped by a suitable modality "!" (read "of course"). Such a modality provides a subtle tool for investigating the sharable nature of objects: $!!(A)$ is by no means isomorphic to $!(A)$. Thus, we immediately have a stratified universe that is the logical counterpart of Lamping's levels. In particular, the croissant and the bracket of Lamping are naturally and respectively associated with the two operations $\epsilon : !(A) \to A$ and $\delta : !(A) \to !!(A)$ of the comonad "!" of Linear Logic. According to such a correspondence, the optimal rewriting rules can then be understood as a "local implementation" of naturality laws, that is as the broadcasting of some information from the output to the inputs of a term, following its connected structure. In order to get optimal reductions, we just have to pursue this propagation of natural transformations as far as it is required to put in evidence new redexes. This can be simply achieved by keeping a subset of the full broadcasting system. The other rules still play a central role in the read-back phase.

4.1 Intuitionistic Linear Logic

In this section we shall rapidly survey the main notions of Intuitionistic (Implicative) Linear Logic. For the moment, we shall merely introduce the "truly" linear part of the logic, postponing the discussion of the "of course" modality to section 4.2.

Intuitionistic Linear Logic is the logical system defined by the following rules:

$$(Exchange)\ \frac{,x:A,y:B,\Delta \vdash M:C}{,y:B,x:A,\Delta \vdash M:C}$$

$$(Axiom)\ \frac{}{x:A \vdash x:A} \qquad (Cut)\ \frac{\vdash M:A \quad \Delta,x:A \vdash N:B}{,\Delta \vdash N[{}^{M}/_{x}]:B}$$

$$(\multimap,l)\ \frac{\vdash M:A \qquad x:B,\Delta \vdash N:C}{,y:A \multimap B,\Delta \vdash N[{}^{(y\ M)}/_{x}]:C} \qquad (\multimap,r)\ \frac{,x:A \vdash M:B}{\vdash \lambda x.M:A \multimap B}$$

It is readily seen that this fragment of Linear Logic corresponds with linear λ-calculus†, via the Curry-Howard analogy (see also Exercise 4.2.1).

Linear Logic proofs can be nicely represented by means of *Proof Nets* [Gir87, Dan90]. Proof Nets provide a graphical representation that eliminates most of the syntactical bureaucracy due to the arbitrary sequentialization of inference rules in a linear or tree-like representation of proofs. In a sense, they are an "abstract syntax" for proofs, compared with the "concrete syntax" of usual representations.

Fig. 4.1. Proof Net links.

Proof Net links are depicted in Figure 4.1. A proof of a sequent $A_1,\ldots,A_n \vdash B$ is represented as a net with $n+1$ distinguished nodes, called "conclusions": n negative conclusions for the hypothesis $A_1,\ldots,A_n$, and one positive conclusion for B.

Definition 4.1.1 (proof net) Proof Nets are inductively defined as follows:

† A λ-term is linear when it contains exactly one occurrence of each variable occurring free in it, and any of its subterms is linear. In linear λ-calculus all the terms are linear.

- Every axiom link is a proof net. Both the nodes are *conclusions* of the proof net.

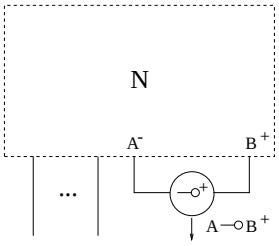


Fig. 4.2. Positive linear implication.

- Let N be a proof net. If A is a negative conclusion and B is the positive conclusions of N, then the net in Figure 4.2 is a proof net. All conclusions of N but A and B are negative conclusions of the new proof net. A ⊸ B is the new positive conclusion.

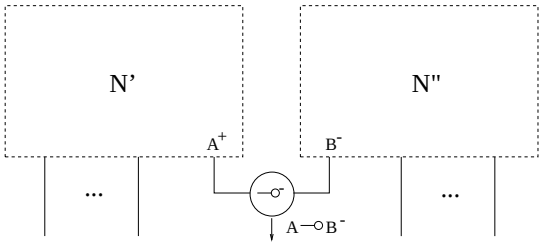


Fig. 4.3. Negative linear implication.

- Let N′ and N″ be proof nets. If A is the positive conclusion of N and B is a negative conclusion of N″, then the net in Figure 4.3 is a proof net. All conclusions of N′ and N″ but A and B are conclusions of the new proof net. A ⊸ B is a new negative conclusion. The unique positive conclusion is the positive conclusion of N″.
- Let N′ and N″ be proof nets. If A is the positive conclusion of N′ and A is a negative conclusion of N′, then the net in Figure 4.4 is a proof net. All conclusions of N′ and N″ but A and B are conclusions of the new proof net. The unique positive conclusion is the positive conclusion of N″.

The cut-elimination rules are given in Figure 4.5.

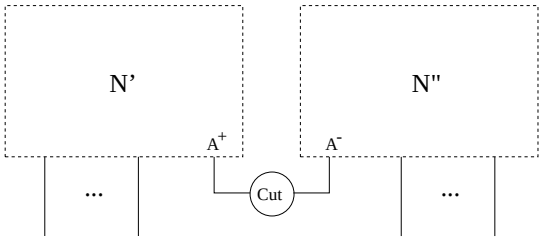


Fig. 4.4. Cut.

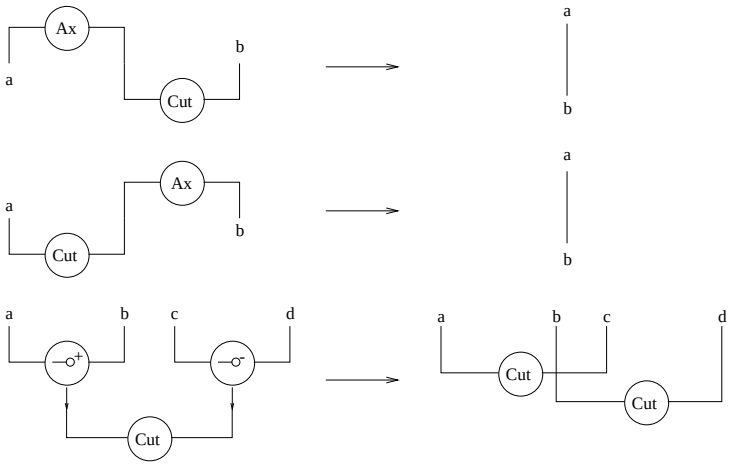


Fig. 4.5. Proof Net cut-elimination.

By the Curry-Howard correspondence, we can use Proof Nets to represent linear λ-terms. In particular, a linear λ-term M with free variables $x_1, \ldots, x_n$ is translated in a proof net M^* with $n+1$ conclusions: the n negative conclusions correspond to the free variables of M; the unique positive conclusion is the "root" of the term.

Example 4.1.2 The λ-term $\lambda x.\, \lambda y.\, ((\lambda z.\, z\ x)\ y)$ is represented by the proof net of Figure 4.6

The reader should not be scared by the net of Figure 4.6. Rearranging the graph according to a different topological setting, it is easy to recognize the usual abstract syntax tree (see Figure 4.7).

A simple procedure for transforming an abstract syntax tree of a λ-term into a proof net is the following:

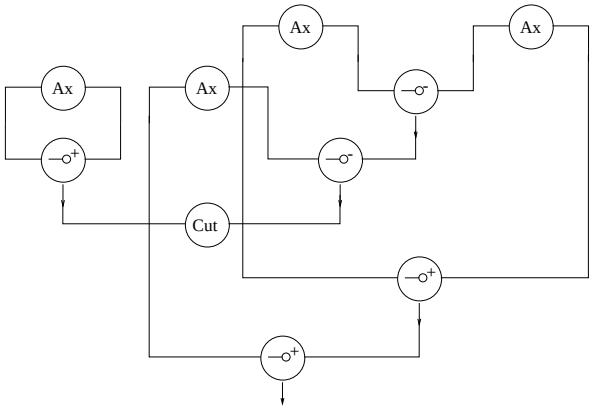


Fig. 4.6. The proof net of $\lambda x.\lambda y.((\lambda z.\, z\ x)\ y)$.

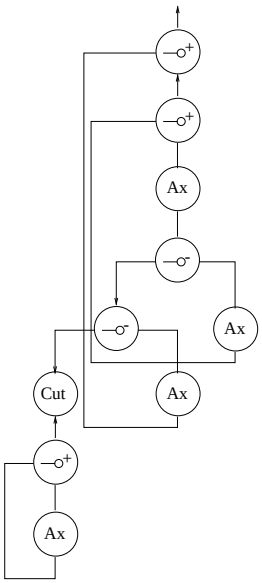


Fig. 4.7. Rearranging the proof net of $\lambda x.\lambda y.((\lambda z.\, z\ x)\ y)$.

(i) Change each application node in a $\multimap^-$ node, and each λ node in a $\multimap^+$ node;

(ii) For every variable x add an axiom link leading to a new node $x^\perp$.

(iii) If a variable x is bound, add a link from $x^\perp$ to the node corresponding to its λ-binder.

(iv) Given an application $(M\ N)$, add an axiom link before the application and a cut link between M and the application.
(v) Remove all cut-axiom pairs introduced by this procedure (typically, in applicative chains of the kind $(M\ N_1\ \ldots\ N_k)$).

Cut and axioms links are a sort of identity connections whose purpose is to preserve the orientation of the net—in all the previous drawings of proof nets but Figure 4.7 the edges should be intended oriented top-down. Hence, it is readily seen that cut and axioms links are not really necessary: they can be safely replaced by wires once we assume that the edges are not oriented.

Even in this linear setting, we already have some important relations between proof nets and sharing graphs. Namely:

(i) The explicit connection between bound variables and their respective binders;
(ii) The cut-elimination rule for $\multimap$, that is exactly equivalent to the β-reduction rule of sharing graphs.

4.2 The "!" modality

In order to exit from the linear framework, and to recover the full power of intuitionistic logical systems, Linear Logic is equipped by a suitable modality "!" (read *of course*). The modality applies to formulae, expressing their non linear nature. So, a formula of the kind $!(A)$ can be freely duplicated and erased. From the logical point of view, this means that we allow contraction and weakening on these formulae:

$$(\mathit{Contraction})\ \frac{x : !(A), y : !(A), \ \vdash M : B}{z : !(A), \ \vdash M[^{z}/_{x}, ^{z}/_{y}] : B}$$

$$(\mathit{Weakening})\ \frac{\vdash M : B}{x : !(A), \ \vdash M : B}$$

The modality "!" can be introduced on the left-hand side of a sequent by the following dereliction rule:

$$(\epsilon)\ \frac{x : A, \ \vdash M : B}{x : !(A), \ \vdash M : B}$$

For introducing "!" on the right, we must ensure that all the hypotheses on the left-hand side of the sequent have been previously derelicted

(if $\Gamma = A_1, \ldots, A_n$, we shall write $!\Gamma$ for $!(A_1), \ldots, !(A_n)$):

$$(!, r)\ \frac{!\Gamma \vdash M : B}{!\Gamma \vdash M : !B}$$

Intuitionistic Logic is fully recovered via the isomorphism $A \to B = !(A) \multimap B$.

Exercise 4.2.1 The following inference rules plus Exchange, Axiom and Cut define the implicative fragment of intuitionistic logic (IL):

$$(\to, l)\ \frac{\Gamma \vdash M : A \qquad x : B, \Delta \vdash N : C}{y : A \to B, \Delta \vdash N[(y\ M)/x] : C} \qquad (\to, r)\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

$$(\textit{Contraction})\ \frac{x : A, y : A, \Gamma \vdash M : B}{z : A, \Gamma \vdash M[z/x, z/y] : B} \qquad (\textit{Weakening})\ \frac{\Gamma \vdash M : B}{x : A, \Gamma \vdash M : B}$$

(i) Exploiting the translation $A \to B = !(A) \multimap B$, prove that for any IL proof there is a corresponding Linear Logic proof ending with the corresponding sequent.

(ii) Find another translation of IL into Linear Logic. (*Hint:* What would happen if every formula would be prefixed by an "!" modality?)

In order to have a better correspondence between sharing graphs and linear logic, it is convenient to split $(!, r)$ in two more elementary steps:

$$(!)\ \frac{\Gamma \vdash M : B}{!\Gamma \vdash M : !(B)}$$

$$(\delta)\ \frac{x : !(!(A)), \Gamma \vdash M : B}{x : !(A), \Gamma \vdash M : B}$$

The two formulations of the "!" modality we have given are equivalent. In fact, it is immediate that $(!)$ and (δ) imply $(!, r)$, and that $(!, r)$ imply $(!)$ (note that to prove the latter implication we also use (ϵ)). Hence, the only think we have left to prove is that $(!, r)$ implies (δ). For this purpose, let us note that $!(A) \vdash !(!(A))$ is provable in the first formulation; then, by a cut (which can be eventually eliminated), we have:

$$\frac{!(A) \vdash !(!(A)) \qquad !(!(A)), \Gamma \vdash B}{!(A), \Gamma \vdash B}$$

4.2.1 Boxes

The proof net notation for contraction, weakening, (ϵ) and (δ) is straightforward: we just introduce four new links of arity three, one, two, and two, respectively (see Figure 4.8). As we shall see, these links have some clear operational analogies with the corresponding sharing graphs operators.

Fig. 4.8. Sharing graph nodes for contraction, weakening, (ϵ) and (δ).

Less obvious is the proof net notation for (!); Girard's solution was that of using *boxes*. A box denotes a datum which has a non-linear use. In particular, data inside boxes may be duplicated and erased. The role of the box is merely to stress the character of *global entity* of the data it contains. No direct interaction between the external and the internal world of a box is allowed. We can duplicate a box, erase it, open it, build a new box around another one, or merge two boxes together. All these operations are described by the rewriting rules of Figure 4.9.

A more suggestive solution for representing boxes is that of using different levels, in a three dimensional space. In this case, "building a box" around a proof net N (i.e., applying the (!) rule), means to shift N at a higher level (in practice, we can use integers to denote the level of links, as in sharing graphs)

In Figure 4.10 you will find the cut elimination rules for boxes, rephrased in the "three dimensional" approach. We also removed the (inessential) cut-link, and adopted the convention of writing the unique positive conclusion of the box at its top. Note that the last rule of Figure 4.9 can now be omitted (two regions at the same level can be seen as a unique region).

The relation with sharing graphs should be now clear: in both cases, the effect of croissants and brackets is just that of shifting links at a lower or higher level. The only difference is that this operation is considered as a unique global move (i.e. over a box) in the cut-elimination process, while it has been decomposed in more atomic steps in the optimal implementation (we shall come back to this point in the next section).

The analogy between proof nets of Linear Logic and sharing graphs is particularly striking when considering the proof-net encoding of λ-

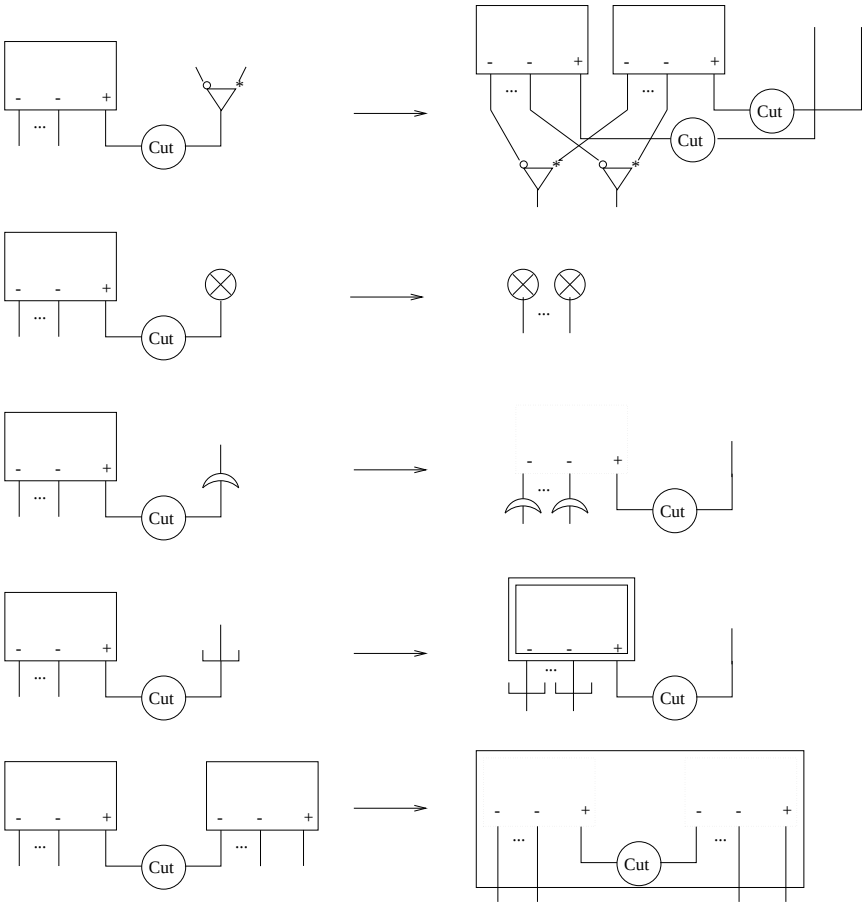


Fig. 4.9. Rewriting of boxes.

terms as provided by the Curry-Howard correspondence. In the "three dimensional" approach, this is given by the rules in Figure 4.11. This is exactly the same as the encoding of λ-terms in sharing graphs given in Figure 3.13.

4.2.2 A Remark for Categoricians

The previous exposition of Intuitionistic Linear Logic is particularly close to its categorical interpretation [BBdPH92]. The general ideas are the following:

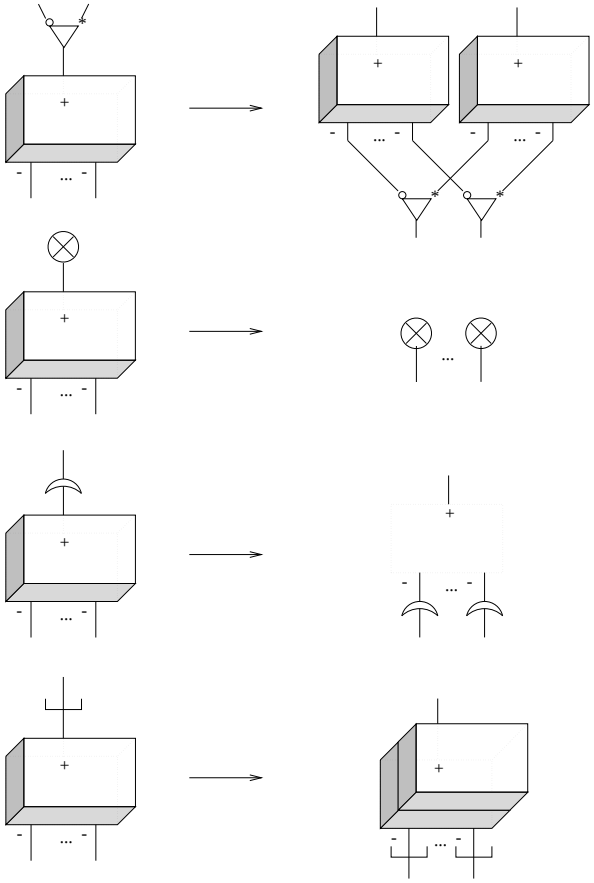


Fig. 4.10. Three dimensional interpretation of boxes.

- ! is a comonad, and it is thus equipped with two natural transformations $\epsilon : ! \to I$ (where I is the identity) and $\delta : ! \to !!$. Explicitly, the naturality of ϵ and δ is expressed by the following equations: given an arrow $f : !(A) \to !(B)$ (i.e., a "proof" f inside a box), we have:

$$\epsilon_A \circ !(f) \to f \circ \epsilon_B$$
$$\delta_A \circ !(f) \to !(!(f)) \circ \epsilon_B$$

- Each object $!(A)$ is a comonoid, and it is thus equipped with two morphisms $E_A : !(A) \to e$, (where e is the unit of the monoidal category)

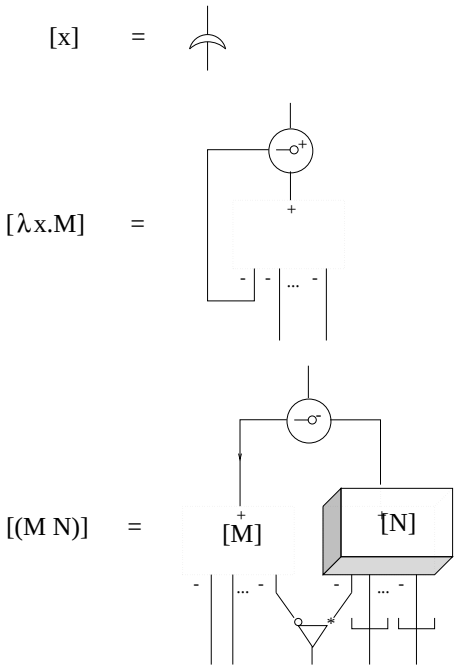


Fig. 4.11. Translation of λ-terms into "three dimensional" sharing graphs.

and $\Delta_A : !(A) \to !(A) \otimes !(A)$. Again, these morphisms are natural:

$$E_A \circ !(f) \to id_e \circ \epsilon_B$$
$$\Delta_A \circ !(f) \to !(f) \otimes !(f) \circ \Delta_B$$

- ! behaves as a monoidal multi-functor, that is, we have a natural transformation $m_{A,B} : !A \otimes !B \to !(A \otimes B)$ such that, given $f : A \to C, g : B \to D$, then

$$m_{C,D} \circ !f \otimes !g = !(f \otimes g) \circ m_{A,B}$$

- ϵ and δ are *monoidal* natural transformations.

According to the previous ideas, the rule $(!, r)$ has then the following interpretation: given an arrow $g : !A \otimes !B \to C$, we get the new arrow $!(g) \circ m_{!A,!B} \circ \delta_A \otimes \delta_B : !A \otimes !B \to !C$.

The interesting fact is that the morphism m is *implicit* in the proof net representation, that is essentially justified by the following equations:

$$\epsilon_{A \otimes B} \circ m = \epsilon_A \otimes \epsilon_B \tag{4.1}$$

$$\delta_{A\otimes B} \circ m = !(m) \circ m \circ \delta_A \otimes \delta_B \quad (4.2)$$

$$E_{A\otimes B} \circ m = E_A \otimes E_B \quad (4.3)$$

$$\Delta_{A\otimes B} \circ m = m \otimes m \circ \alpha \circ \delta_A \otimes \delta_B \quad (4.4)$$

where $\alpha : (!A \otimes !A) \otimes (!B \otimes !B) \cong (!A \otimes !B) \otimes (!A \otimes !B)$ in (4) is the unique structural isomorphism.

Equations (1) and (2) follows from the hypotheses that ϵ and δ are monoidal natural transformations, and equations (3) and (4) from the good interplay between the comonad $(!, \epsilon, \delta)$ and the comonoids $(!A, E, \Delta)$.

The rewriting rules of Figure 4.10 are just the naturality laws of ϵ, δ, Δ and E up to the implicit morphism m. For instance, given $g : !A \otimes !B \to C$ and the "box" $!(g) \circ m_{!A,!B} : !A \otimes !B \to !C$, we have:

$$\begin{aligned} \delta_C \circ !(g) \circ m_{!A,!B} &= !(!(g)) \circ \delta_{!(!A\otimes !B)} \circ m_{!A,!B} \\ &= !(!(g)) \circ !(m) \circ m \circ \delta_{!A} \otimes \delta_{!B} \\ &= !(!(g) \circ m) \circ m \circ \delta_{!A} \otimes \delta_{!B} \end{aligned}$$

You will also note that the reduction system does not have any counterpart for the three comonad equations of "!", namely:

$$\begin{aligned} \epsilon_{!A} \circ \delta_A &= Id_{!A} \\ !(\epsilon_A) \circ \delta_A &= Id_{!A} \\ !(\delta_A) \circ \delta_A &= \delta_{!(A)} \circ \delta_A \end{aligned}$$

These rules are not essential to pursue β-reduction, since we may always propagate the natural transformations in order to put in evidence new redexes. However, these natural transformations will eventually accumulate over free variables.

This is actually a major problem of optimal reduction too: the so-called dynamical accumulation of square brackets and croissants. We shall come back in more details to this problem in Chapter 9.

4.3 The duplication problem

In the rules of Figure 4.10, the box $!(\,.\,)$ is considered as a single, global entity. In particular, the reduction of Figure 4.12 amounts to physically duplicate the whole box. If we had a β-redex inside the box, this would be eventually duplicated, loosing the possibility to share its reduction.

As we know, in order to get optimal reduction, we are forced to proceed to this duplication in a sort of "lazy" way. In other words, the duplication operator Δ at the output of the box must be propagated

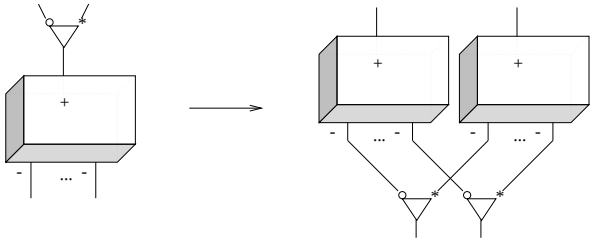


Fig. 4.12. Duplication.

towards the inputs travelling *inside* the box. Since the graph inside the box is a connected structure, the obvious idea for performing this propagation is that of doing a "full broadcasting" of Δ along all links in the box; if two copies of Δ meet each other they annihilates, otherwise the broadcasting stops when the operator reaches a link at its own level. In this way, part of the box may be shared by different terms, and we are forced to implement *all* the reduction rules of Figure 4.10, following the same local strategy as for Δ.

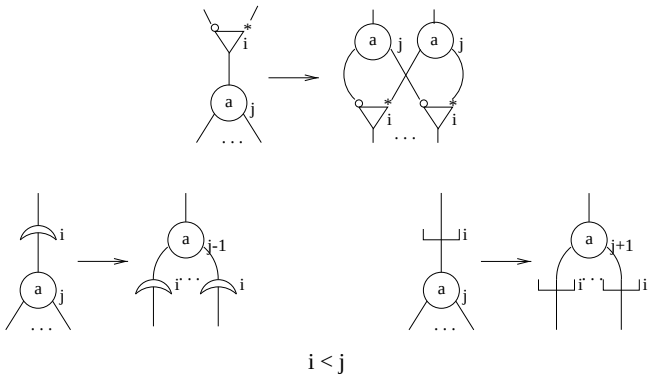


Fig. 4.13. Broadcasting rules.

Every operator will act over the link l it traverses according to its semantics: the duplicator (fan) will duplicate l, the croissant will decrement the level of l, the square bracket will increment it (and the garbage will collect l). Intuitively, since all operators travel in the same direction and according to a same strategy, there is no possible confusion in this process. So, the rules in Figure 4.13 are all semantically correct, no matter at which port of a the operator of lower index is interacting

to. A proof of the soundness of these rules will be given in Chapter 7. Besides, we already know that in order to achieve optimal reduction, it is enough to limit the possibility of reduction to the case of interaction at the *principal ports* of the nodes. Nevertheless, the fact that all broadcasting rules are correct, has another important consequence: in order to "read back" a term obtained by an optimal sharing reduction, we could just apply all the remaining broadcasting rules, as we will see in detail in Chpater 7.

5
Redex Families and Optimality

In 1978, Lévy introduced the notion of *redex family* in the λ-calculus with the aim to formally capture an *intuitive* idea of *optimal sharing* between "copies" of a same redex. In order to comfort his notion of family, Lévy proposed several alternative definitions inspired by different perspectives and proved their equivalence.

The most abstract approach to the notion of family (see [Lév78, Lév80]) is the so called *zig-zag* relation. In this case, duplication of redexes is formalized as residuals modulo permutations. In particular, a redex u with history σ (notation σu) is a *copy* of a redex v with history ρ iff $\rho v \leq \sigma u$ (i.e., there exists τ such that $\sigma = \rho\tau$ up to a notion of permutation of redexes, and u is a residual of v after τ). The family relation $\simeq$ is then the symmetric and transitive closure of the copy relation. Now, let us draw reduction arrows downwards. Pictorially, the family reduction gives rise to an alternate sequence of descending and ascending reduction arrows. This is the reason why it is also known as "zig-zag" relation.

Another approach is that of considering the *causal history* of redexes. Intuitively, two redexes can be "shared" if and only if they have been "created in the same way" (or, better, their *causes* are the same). This is formalized by defining an *extraction relation* over redexes (with history) σu, which throws away all the redexes in σ that have not been relevant for the creation of u. The canonical form we obtain at the end of this process essentially expresses the causal dependencies of u along the derivation (we may deal with causal *chains* instead of *partial orders* since *only standard derivations* are considered).

The most "operational" approach to the family relation is based on a suitable labeled variant of the λ-calculus [Lév78]. The idea of labels is essentially that of marking the "points of contact" created by reductions.

In particular, labels grow along the reduction, keeping a trace of its history. Two redexes are in a same family if and only if their labels are identical.

The equivalence between zig-zag and extraction is not particularly problematic (see [Lév80]). On the contrary, the proof of the equivalence between extraction (or zig-zag) and labeling is much more difficult (see for instance the long proof that Lévy gave in its thesis [Lév78]). For this reason, we shall postpone its proof in Chapter 6, after the introduction of the so-called *legal paths*. In fact, given a term T, legal paths will give us a complete characterization in terms of paths (virtual redexes) in T of the redexes generated along the reduction of T. Intuitively, a legal path for a redex R will be a suitable composition of subpaths connecting those redexes required for the creation of R. Any legal path φ will correspond to a reduction ρ composed of redexes in φ only; moreover, the redex with history ρR will be in normal form with respect to the extraction relation. The main issue of section 6.2 of Chapter 6, will be to prove that the labels generated along the reduction of T "are" indeed legal paths in T, and vice versa.

5.1 Zig-Zag

Let $\Delta = \lambda x.(x\,x)$, $F = \lambda x.(x\,y)$, $I = \lambda x.\,x$. Consider the possible reductions of $M = (\Delta\,(F\,I))$, represented in Figure 5.1.

Intuitively, there are just three kinds of redexes in this example: R, S and T. In the case of R and S, the common nature of these redexes is clear, for all R_i and S_i are residuals of R and S. The case of T is more complex. In fact, T, T_1 and T_2 are not residuals of any redex in M: they are just created by the immediately previous reduction. The two redexes T_1 and T_2 clearly look sharable (and they are indeed "shared" as the unique redex T in the innermost reduction on the left). However, the only way to establish some formal relation between these redexes is by closing residuals downwards. In fact, T_1 and T have a common residual T_3 inside the term $((I\,y)\,(I\,y))$; similarly, T_2 and T have a common residual T_4 in the same term. By transitive closure, we can thus conclude the "common nature" of T_1 and T_2: in a sense, T_1 is "the same" redex as T_3, which is the same as T, which in turn is the same as T_4 which is the same as T_2. So, the idea of "closing residuals downwards" seems the right extension allowing to connect redexes with a common origin, but without any common ancestor.

The formalization of the previous intuition requires however some

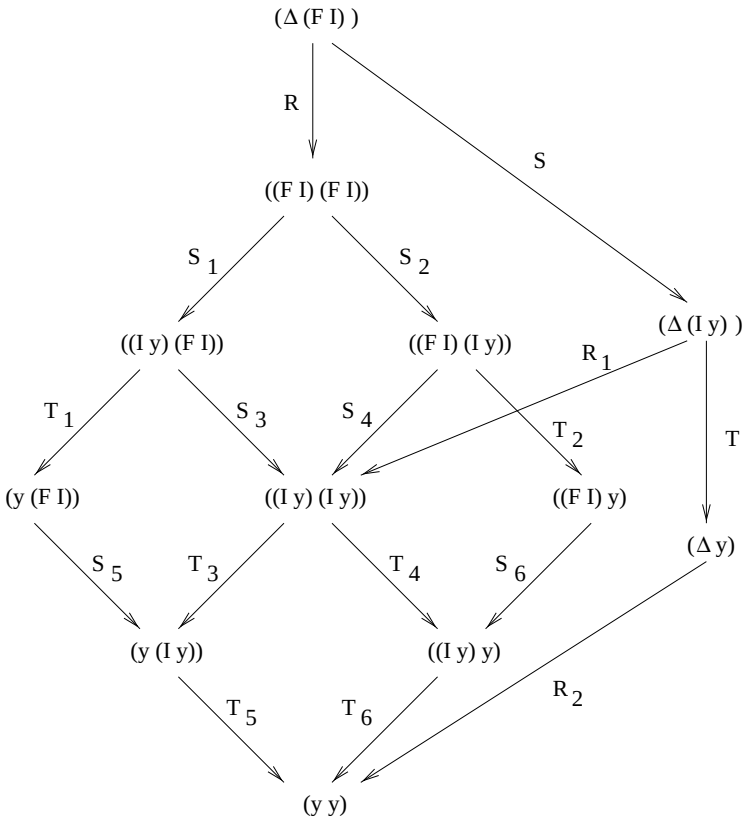


Fig. 5.1. $\Delta = \lambda x.\,(x\,x), F = \lambda x.\,(x\,y), I = \lambda x.\,x$

care. In particular, let us start noticing that T_1 and T_2 should not be connected if the initial expression would be $((F\,I)\,(F\,I))$, instead of M. The intuitive reason is that, while we want to preserve the sharing "inherent" in the initial λ-term, we are not interested in recognizing common subexpressions generated along the reduction—the equivalence of such new subexpressions should be considered as a mere syntactical coincidence. For instance, although that reducing any of the two redexes in $(I(I\,x))$ we get the "same" term, the term $(I\,x)$ obtained reducing the outermost redex and the term $(I\,x)$ obtained reducing the innermost redex should be considered distinct. The reasons should be computationally clear: to look for common sharable subexpressions would be too expensive in any practical implementation. Furthermore, even though we could imagine an optimization step based on a preprocessing rec-

ognizing common subexpressions in the initial term, to run such an optimizing algorithm at run-time would be infeasible.

A main consequence of the latter assumption is that the relation we are looking for will not really be a relation over redexes, it will be rather relativized with respect to the reduction of some initial expression. In other words, we shall have to consider pairs composed of a redex R together with the reduction ρ that generated it. Namely, given a derivation $T \xrightarrow{\rho} T' \xrightarrow{R}$, we shall say that ρ is the *history* of the redex R in T'. As a consequence, any redex with history ρR determines a unique initial term T with respect to which we are considering sharable redexes, and two redexes with history will be comparable only when they start at the same initial term.

Let us come back to the see how the previous considerations apply to our previous examples. Since we relativized redexes with respect to an initial expression and we equipped them with an history, redexes are no more identified only by their syntactical position in the term. For instance, the term $M = (I\,(I\,x))$ has two redexes R and S. By firing any of them we obtain the term $(I\,x)$, in which we have a unique redex T. Since T is a residual of both R and S we could be tempted to conclude that R and S are connected by our "sharing" relation. On the contrary, for we do not want to relate distinct redexes in the initial term, the equivalence of the result reducing R or S is merely incidental. In fact, introducing histories, the two reductions respectively give RT and ST. Thus, the two redexes RT and ST could be related via our sharing relation, only if their histories R and S could be related via such a relation. That is, only if M would be some partial result of the computation of another term N, and we could find two reductions ρ and σ of N such that ρR and σS might be equated applying zig-zag or the extraction relation. This is indeed the case for the term $((I\,y)\,(I\,y))$ generated along the reduction in Figure 5.1. The two redexes T_3 and T_4 do not have any common ancestor. Nevertheless, zig-zag will relate them (see Example 5.1.9) and the extraction relation will associate to any ρT_3 and any σT_4 the same redex with history ST (see Example 5.2.3 and Exercise 5.2.5).

5.1.1 Permutation equivalence

Before to give the definition of families, we present some standard results of λ-calculus. For an unabridged presentation of λ-calculus and for the proof of the previous results, we refer the reader to Hindley and Seldin's book [HS86], even if the most complete source for the syntax properties

of λ-calculus is definitely Barendregt's book [Bar84]. Nevertheless, especially for readers without a good familiarity of λ-calculus, we suggest to start with Hindley and Seldin. We also remark that the proofs of the following results are actually subsumed by the results on the labeled calculus that we will give in section 5.3.

The first notions we need are the ones of *residual* of a redex (note that in the proof of correctness we have already met this concept using graphs) and of *permutation* of a reduction. To introduce them, let us recollect what we did learn by analyzing the example of $(I(Iy))$. The relevant point was that the syntactical equivalence of the redexes RT (T with history R) and ST (T with history S) is incidental, for the redex T in (Iy) is respectively a residual of S w.r.t. R and a residual of R w.r.t. S. Abstracting from the contingent syntax of λ-calculus that equates the result of the two reductions above, there is clearly no reason to equate RT and ST.

The previous point is settled introducing a permutation operation on redexes such that two reductions $T \stackrel{\rho}{\twoheadrightarrow} T'$ and $T \stackrel{\sigma}{\twoheadrightarrow} T'$ are permutation equivalent only when ρ is a suitable permutation of the redexes in σ. By the way, in this framework the word permutation must be interpreted in a wide sense, for the contraction of a redex might cause the duplication of another redex following it; thus, permuting a reduction, its length might shrink or expand. In the previous example, since R and S are distinct redexes of the initial term, the corresponding reductions cannot be equated by permutation. Furthermore, in such a formal setting we should not use the name T to denote the redex of (Iy); taking into account that reducing R the redex T is what remains of the redex S, while reducing S the redex T is what remains of R, we should rather use the names S/R (the residual of S after R) and S/R. It is then intuitive that in order to get two equivalent reductions, the reductions R and S should be completed reducing S/R and R/S, respectively. Namely, while RS/R and SR/S will not be equated as redexes with history, the two reductions RS/R and SR/S are obviously equivalent by permutation.

To denote the residual of a redex, let us assume that we can mark some redexes of a λ-term underlining them and associating a name to each underlining. For instance, let us consider again the term of Figure 5.1

$$\underline{(\Delta\underline{(FI)}_S)}_R$$

we have underlined the redexes R and S of $(\Delta\,(F\,I))$ using the names R and S (say that the redexes are underlined by R and S, respectively).

Let us assume that β-reduction preserves marking and naming. It is intuitive that, after a reduction, what is left of a redex marked by a name R, say its residual, is formed of all the redexes of the result with the same marking R.

Example 5.1.1 For instance,

$$\underline{(\Delta\,\underline{(F\,I)}_{S})}_{R} \xrightarrow{R} (\underline{(F\,I)}_{S}\,\underline{(F\,I)}_{S}) \xrightarrow{S_1} ((I\,y)\,\underline{(F\,I)}_{S}) \xrightarrow{S_3} ((I\,y)\,(I\,y))$$

is a marked reduction taken from the example in Figure 5.1.

The previous example also shows that when a marked redex is fired there is no trace of its underlining in the result, which corresponds to the fact that after its contraction a redex has no residual.

Definition 5.1.2 (residual) Let $\rho : M \twoheadrightarrow N$. Let us assume that R is the only redex of M marked by an underlining with name R. The set R/ρ of the redexes of N marked by an underlining with name R is the *residual* of R under ρ. The redex R is the *ancestor* of any redex $R' \in R/\rho$, while any of these R' is a *residual redex* of R.

The previous definition gives an example of how to use marking. In general, given a reduction $\rho : M \twoheadrightarrow N$, we associate to ρ a corresponding marked reduction assuming that each redex of M is underlined using its name. As a consequence, two redexes R and S of M will be always underlined using distinct names. Let us however remark that this is not true anymore for the result of a reduction (see the example above).

Let $\mathcal{F}$ be a set of redexes of a λ-term M. A reduction $\rho : M \twoheadrightarrow N$ is *relative to* $\mathcal{F}$ when no redex underlined by a name $R \notin \mathcal{F}$ is reduced along the marked reduction corresponding to ρ. For instance, the reduction in Example 5.1.1 is relative to $\{R, S\}$.

A reduction $\rho : M \twoheadrightarrow N$ is a (complete) *development* of a set of redexes $\mathcal{F}$ of M when ρ is relative to $\mathcal{F}$ and, after the marked reduction corresponding to ρ, N does not contain any redex underlined by a name $R \in \mathcal{F}$.

Theorem 5.1.3 (finite developments) *Let $\mathcal{F}$ be a set of redexes of a λ-term.*

(i) *There is no infinite reduction relative to $\mathcal{F}$.*

(ii) *All developments end at the same term.*

(iii) *For any redex* R *of* M *and any pair of developments* ρ *and* σ *relative to* $\mathcal{F}$*, we have that* $R/\rho = R/\sigma$.

A first consequence of the previous lemma is that there is no ambiguity in writing $\mathcal{F} : M \to N$, for we may assume that this is a notation for all the developments of $\mathcal{F}$. Besides, we get in this way a parallel reduction of λ-terms in which a whole set of redexes is simultaneously reduced at each step. Thus, the composition $\mathcal{F}_1\mathcal{F}_2\cdots\mathcal{F}_k$ denotes a sequence of parallel reduction steps $\mathcal{F}_i : M_{i-1} \to M_i$. In particular, as there is no restriction on the shape of the previous sets, $\mathcal{F}_i$ might even be empty; in which case $M_{i-i} = M_i$. In spite of this, we stress that $\emptyset \neq \epsilon$ (being ϵ the empty reduction), then in the equivalence by permutation that we will later define, we will have to explicitly state that the empty reduction and the reduction relative to an empty set of redexes are equivalent.

The notion of residual can be directly extended to parallel reductions, i.e., $R/\mathcal{F}$ is the residual of the redex R under (any development of) $\mathcal{F}$ (let us note that by Theorem 5.1.3 such a definition is not ambiguous). Furthermore, residuals can be also extended to set of redexes, taking $R' \in \mathcal{G}/\mathcal{F}$ iff $R' \in R/\mathcal{F}$ for some $R \in \mathcal{G}$.

Of particular relevance is the reduction $\mathcal{F} \sqcup \mathcal{G} = \mathcal{F}(\mathcal{G}/\mathcal{F})$, that is, given two sets of redexes $\mathcal{F}$ and $\mathcal{G}$ of a λ-term, the reduction obtained reducing first the set $\mathcal{F}$ and then the residual of $\mathcal{G}$. In terms of developments, it is not difficult to realize that, appending a development σ of $\mathcal{G}/\mathcal{F}$ to a development ρ of $\mathcal{F}$, we get a development $\rho\sigma$ of $\mathcal{F} \cup \mathcal{G}$. Nevertheless, $\mathcal{F} \cup \mathcal{G}$ and $\mathcal{F} \sqcup \mathcal{G}$ differ, for they denote different sets of reduction sequences. Hence, assuming parallel reduction as atomic, we are not allowed to equate them. What we expect to equate are instead reductions in which redexes are permuted preserving their grouping, i.e., like in the two reductions $\mathcal{F} \sqcup \mathcal{G}$ and $\mathcal{G} \sqcup \mathcal{F}$.

Lemma 5.1.4 (parallel moves) *Let* $\mathcal{F}$ *and* $\mathcal{G}$ *be set of redexes of a* λ*-term* M*. We have that:*

(i) $\mathcal{F} \sqcup \mathcal{G}$ *and* $\mathcal{G} \sqcup \mathcal{F}$ *ends at the same expression;*

(ii) $\mathcal{H}/(\mathcal{F} \sqcup \mathcal{G}) = \mathcal{H}/(\mathcal{G} \sqcup \mathcal{F})$.

The main consequence of the previous lemma is that the equivalence $\mathcal{F} \sqcup \mathcal{G} \equiv \mathcal{G} \sqcup \mathcal{F}$ is sound. We can thus use it as the core of the equivalence of reduction by permutation, the other equations being the ones induced by the fact that we want to get a congruence with respect to composition of reductions.

Definition 5.1.5 (permutation equivalence) The *permutation equivalence* of reductions $\equiv$ is the smallest congruence with respect to composition of reductions satisfying the parallel moves lemma and elimination of empty steps. Namely, $\equiv$ is the smallest equivalence such that:

(i) $\mathcal{F} \sqcup \mathcal{G} \equiv \mathcal{G} \sqcup \mathcal{F}$, when $\mathcal{F}$ and $\mathcal{G}$ are set of redexes of the same λ-term;

(ii) $\emptyset = \epsilon$;

(iii) $\rho\sigma\tau \equiv \rho\sigma'\tau$, when $\sigma \equiv \sigma'$.

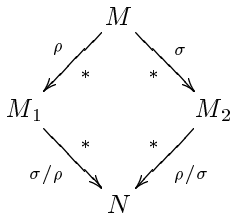
The notion of residual extends to reductions too. In fact, we have the following definition by induction on the length of the reduction:

$$\begin{aligned} \epsilon/\rho &= \epsilon \\ (\sigma\mathcal{F})/\rho &= (\sigma/\rho)(\mathcal{F}/(\rho/\sigma)) \end{aligned}$$

The previous definition allows to conclude this part with the so-called diamond property.

Theorem 5.1.6 (diamond property) *For any pair of reductions* ρ *and* σ *starting from the same term, we have that* $\rho\,(\sigma/\rho) = \sigma\,(\rho/\sigma)$.

That can be graphically depicted by the following diamond:



Let us remark that the previous commuting diagram is indeed the base of the equivalence by permutation. In fact, two reductions ρ and σ are permutation equivalent if and only composing instances of the previous diagram we can get a commuting diagram in which the composition of the external reductions yields ρ and σ.

The diamond property is a strong version of the so-called Church-Rosser (or confluence) property. In fact, it proves the confluence of the calculus, showing at the same time how to complete any pair of reductions ρ and σ in order to get the same result. Hence, an easy corollary of the previous result is the uniqueness of the normal form (if any).

5.1.2 Families of redexes

Definition 5.1.7 (copy) A redex S with history σ is a *copy* of a redex R with history ρ, written $\rho R \leq \sigma S$, if and only if there is a derivation τ such that $\rho\tau$ is permutation equivalent to σ ($\rho\tau \equiv \sigma$) and S is a residual of R with respect to τ ($S \in R/\tau$).

Definition 5.1.8 (family) The symmetric and transitive closure of the copy relation is called the *family* relation, and will be denoted with $\simeq$.

Explicitly, two redexes R and S with respective histories ρ and σ are in the same family ($\rho R \simeq \sigma S$) if and only if there is a finite sequence $\tau_0 T_0, \tau_1 T_1, \ldots, \tau_k T_k$, with $\tau_0 T_0 = \rho R$ and $\tau_k T_k \leq \sigma S$, such that either $\tau_{i-1} T_{i-1} \leq \tau_i T_i$ or $\tau_i T_i \leq \tau_{i-1} T_{i-1}$, for $i = 1, \ldots, k$. That, pictorially, gives rise to a sort of "zig-zag" (see Figure 5.2).

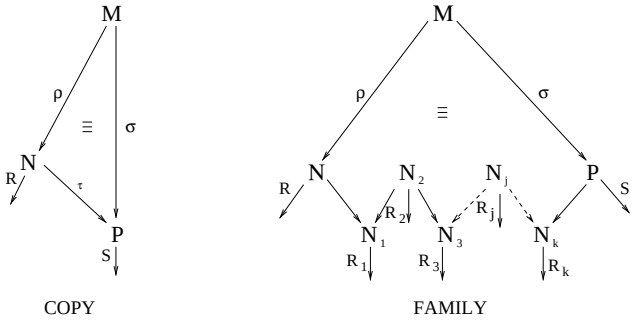


Fig. 5.2. Copy and family relations.

Example 5.1.9 Let us come back to the example of Figure 5.1. We have:

$$RS_1T_1 \leq RS_1S_3T_3 \geq ST \leq RS_2S_4T_4 \geq RS_2T_2$$

From which we conclude that $RS_1T_1 \simeq RS_2T_2$.

Let us now prove a few interesting properties of the copy and family relations.

Lemma 5.1.10 *Let* $\rho \equiv \rho'$ *and* $\sigma \equiv \sigma'$. *Then:*

(i) $\rho R \leq \sigma S$ *iff* $\rho' R \leq \sigma' S$

(ii) $\rho R \simeq \sigma S$ *iff* $\rho' R \simeq \sigma' S$

Proof Obvious, since $\equiv$ is a congruence for composition. □

Although the previous lemma looks straightforward, its relevance is not negligible. Indeed, it says that in order to check that two redexes are in the same family, we can consider any other history of the redexes, provided that they are permutation equivalent to the given ones. In particular, it allows to restrict our analysis to *standard* derivations. In fact, let us remind that a derivation $R_1 R_2 \cdots$ is standard when R_i is not residual of any redex at the left of R_j, for any i and any $j < i$. In the case of parallel reductions, the derivation $\mathcal{F}_1 \mathcal{F}_2 \cdots$ is standard when the previous proviso holds assuming that R_i and R_j are the leftmost-outermost redexes of the respective sets $\mathcal{F}_i$ and $\mathcal{F}_j$. A relevant result of λ-calculus—the so-called *standardization theorem*—ensures that, for any reduction, there exists an equivalent one which is standard.

The permutation equivalence allows to extend the usual preorder given by the prefix relation between reduction sequences. Namely, let us define $\rho \sqsubseteq \sigma$, when $\rho\tau \equiv \sigma$ for some reduction τ.

Lemma 5.1.11 $\rho R \leq \sigma S$ *iff* $\rho \sqsubseteq \sigma$ *and* $S \in R/(\sigma/\rho)$.

Proof ($\Rightarrow$) By definition, if $\rho R \leq \sigma S$ there exists τ such that $\rho\tau \equiv \sigma$ and $S \in R/\tau$. Thus, $\rho \sqsubseteq \sigma$, by definition of $\sqsubseteq$. Moreover, $\tau \equiv \sigma/\rho$, that implies $R/\tau = R/(\sigma/\rho)$ and $S \in R/(\sigma/\rho)$. ($\Leftarrow$) Just take $\tau = (\sigma/\rho)$. □

An interesting consequence of the previous lemma is that the copy relation is decidable.

Lemma 5.1.12 (interpolation) *For any* $\rho \sqsubseteq \sigma \sqsubseteq \tau$ *and* $\rho R \leq \tau T$, *there exists a redex* S *such that* $\rho R \leq \sigma S \leq \tau T$.

Proof By assumption, there exist reductions ρ', σ' and τ' such that $\rho\rho' \equiv \sigma$, $\sigma\sigma' \equiv \tau$, $\rho\tau' \equiv \tau$ and $T \in R/\tau'$. Thus $\rho\tau' \equiv \rho\rho'\sigma'$, and by left-cancellation $\tau' \equiv \rho'\sigma'$. Since R has a residual T after $\tau' \equiv \rho'\sigma'$, then it must also have a residual S after ρ'. □

Lemma 5.1.13 (uniqueness) *If* $\rho R_1 \leq \sigma S$ *and* $\rho R_2 \leq \sigma S$, *then* $R_1 = R_2$.

Proof By the fact that each redex is residual of at most one ancestor. □

Exercise 5.1.14

(i) Prove that $\leq$ is a preorder.

(ii) Prove that $R \simeq \sigma S$ if and only if $S \in R/\sigma$. (*Hint*: For the if part, apply an induction on the definition of $\simeq$, using interpolation and uniqueness.)

5.2 Extraction

Intuitively, two redexes ρR and σS are in a same family if and only if they have been created "in a same way" along σ and ρ. Nevertheless, this intuition is not easy to formalize, since "creation" is a very complex operation in the λ-calculus.

A way to get rid of this complexity is to modify the calculus associating a label to each (sub)term. In this labeled version of the calculus, labels would be a trace of the history of each subterm, and in particular of the way in which each redex has been created. This technique, that generalizes an idea of Vuillemin for recursive program schemes, leads to the *labeled* λ-*calculus* that will be presented in section 5.3. Here, we give an alternative approach to zig-zag that does not involve any modification of the calculus.

Let us assume to have a simplification process (extraction) that for any redex ρR throws away all the redexes in its history ρ that are not relevant to the "creation" of R. At the end of this process, we would essentially obtain the "causal history" of R (with respect to ρ). The causal histories of redexes could then be used to decide when two redexes are in the same family. Namely, we could say that ρR and σS are in the same family when the extraction process contract them to the same redex τT. However, in order to achieve such a strong result, we immediately see that we have to fix some technical details. In particular, when several redexes participate to the creation of R and S, the corresponding casual histories might differ for the order in which such redexes are applied. Unfortunately, this would immediately lead back to permutation equivalence and zig-zag. Thus, as we want a unique "linear" representation of equivalent causal histories (namely, a unique derivation), we are forced to organize redexes in some fixed order. But in view of Lemma 5.1.10, this does not seem a big problem, for restricting to *standard* derivations fulfill the uniqueness requirement and does not force any limitation.

In order to formally define the extraction relation, we need a few preliminary definitions. In the next subsection we will then show that extraction gives indeed a decision procedure for the family relation. In

fact, although the results of section 5.2.1 hold for redexes whose history is in standard form, this does not impact decidability of zig-zag, for there is a recursive algorithm transforming a given reduction ρ into a (unique) standard reduction ρ_s such that $\rho \equiv \rho_s$.

We say that two derivations $\sigma : M \twoheadrightarrow N$ and $\rho : M \twoheadrightarrow P$ are *disjoint* if they contract redexes into disjoint subexpressions of M. This means that $\sigma : M = C[Q_1, Q_2] \twoheadrightarrow C[Q_1', Q_2] = N$, and $\rho : M = C[Q_1, Q_2] \twoheadrightarrow C[Q_1, Q_2'] = P$, for some context $C[\cdot, \cdot]$ with two disjoint holes. We also remind that, in a redex $R = (\lambda x.M\ N)$, the subterm $\lambda x.M$ is the *function part* of R, while M is its *argument part*.

Finally, let us recall that σ/ρ (the residual of a derivation σ with respect to a derivation ρ) is defined inductively by:

$$\sigma/\rho = \begin{cases} \epsilon & \text{when } \sigma = \epsilon \\ (\sigma'/\rho)\,(R/(\rho/\sigma')) & \text{when } \sigma = \sigma' R. \end{cases}$$

where ϵ is the empty reduction.

The next definition gives the only issue that pose some technical difficulties in the definition of extraction. The idea is that, given a redex $R = (\lambda x.\, M\ N)$ and a reduction $R\rho_i$ such that ρ_i works inside the i-th instance N_i of the argument part N of R. Any redex created by $R\rho_i$ could have been created by the direct execution in N of the reduction ρ isomorphic to ρ_i. In order to formalize this point, let us introduce the notation $M^x[\cdot, \cdot, \ldots, \cdot]$ to represent the context obtained by replacing a hole for all the occurrences of a given variable x occurring free in the term M.

Definition 5.2.1 (parallelization) Let $R = (\lambda x.M\ N)$. Let $R\sigma$ be a derivation such that σ is internal to the i-th instance of N in the contractum $M[N/x]$ of R (see the figure in item 4 of Definition 5.2.2). The reduction $\sigma \| R$ (σ *parallelized* by R), is inductively defined as follows:

$$\begin{aligned} \epsilon \,\|\, R &= \epsilon \\ (S\sigma) \,\|\, R &= (S'/R)\,((\sigma/\mathcal{F}) \| (R/S')) \qquad \text{where } S \in S'/R,\ \mathcal{F} = S'/(RS) \end{aligned}$$

Intuitively, for σ corresponds to a reduction applied to the instance of N inserted in the i-th hole of $M^x[\cdot, \ldots, \cdot]$, the reduction σ works on a subterm isomorphic to N. Hence, there exists a reduction

$$\sigma' : C[(\lambda x.\, M\ N)] \twoheadrightarrow C[(\lambda x.\, M\ N')]$$

internal to N and isomorphic to σ, such that

$$\sigma : C[M^x[N, \ldots, N, \ldots, N]] \twoheadrightarrow C[M^x[N, \ldots, N', \ldots, N]].$$

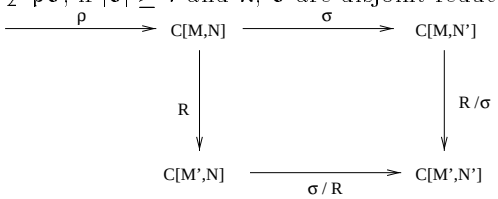
The order of R and σ could then be commuted obtaining $\sigma' \sqcup R$. Nevertheless, it is immediate that after $\sigma' \sqcup R$ all the instances of N in $C[M^x[N,\ldots,N,\ldots,N]]$ are contracted to M'. The reduction $\sigma' \sqcup R$ is indeed the parallelization of σ by R, that is, $\sigma'/R = \sigma' \sqcup R$ (see again item 4 of Definition 5.2.2).

In order to clarify how Definition 5.2.1 fits such an informal idea of parallelization, let us remark that, by induction on $S\sigma$: (i) S' is internal to N; (ii) $\mathcal{F}$ is disjoint from σ; (iii) R/S' is a singleton $\{R'\}$; (iv) $\sigma/\mathcal{F}$ is internal to the i-th instance of the argument of R' in its contractum. By which we conclude the soundness of Definition 5.2.1 and that, as we anticipated, $(S\sigma)\|R = (S'\sigma')/R$, for a suitable reduction σ' internal to N.

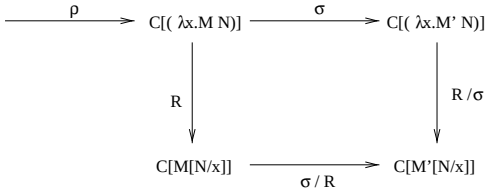
Definition 5.2.2 (extraction) The contraction by extraction $\rhd$ is the union of the following four relations:

(i) $\rho R S \rhd_1 \rho S'$, if $S \in S'/R$;

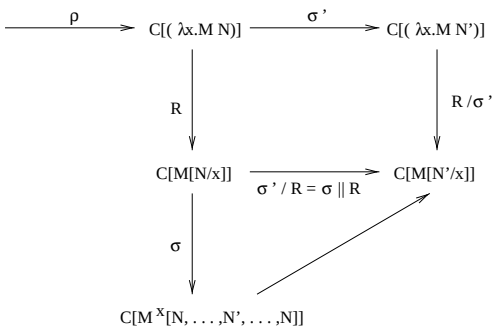
(ii) $\rho\,(R \sqcup \sigma) \rhd_2 \rho\sigma$, if $|\sigma| \geq 1$ and R, σ are disjoint reductions;



(iii) $\rho(R \sqcup \sigma) \rhd_3 \rho\sigma$, if $|\sigma| \geq 1$ and σ is internal to the function part of R;



(iv) $\rho R \sigma \rhd_4^i \rho\sigma'$, if $|\sigma| \geq 1$, σ is internal to the i-th instance of the argument of R in its contractum, and $\sigma'/R = \sigma\|R$.



The *extraction* relation $\trianglerighteq$ is the transitive and reflexive closure of $\rhd$.

Example 5.2.3 Let us consider again the example in Figure 5.1. By parallelization, we have $RS_1T_1 \rhd ST$ and $RS_2T_2 \rhd ST$. By the second rule of the extraction relation, we have $RS_1S_3T_3 \rhd RS_1T_1$ and $RS_2S_4T_4 \rhd RS_2T_2$. Summarizing,

$$RS_1S_3T_3 \trianglerighteq ST \trianglelefteq RS_2S_4T_4.$$

Which is the same result that we could have get applying zig-zag (see Example 5.1.9).

Theorem 5.2.4 ([Lév80]) $\trianglerighteq$ *is confluent and strongly normalizing.*

Proof See [Lév80] or solve Exercise 5.2.6. □

Exercise 5.2.5 Let $\rho_i R_i$, $\sigma_i S_i$, and $\tau_i T_i$ be the redexes with history relative to the example in Figure 5.1. Applying $\trianglerighteq$ to each of them, verify that extraction is confluent and strongly normalizing. Furthermore, check that: (*i*) R is the unique normal form of any $\rho_i R_i$; (*ii*) S is the unique normal form of any $\sigma_i S_i$; (*iii*) ST is the unique normal form of any $\tau_i T_i$.

Exercise 5.2.6 Prove the following fact:

Let R and S be two distinct redexes in a term M. If T, T_1, T_2 are such that $T \in T_1/(R/S)$, $T \in T_2/(S/R)$, then there exists some redex T' such that $T \in T'/(R \sqcup S)$ and $T \in T'/(S \sqcup R)$.

Use the previous result to prove Theorem 5.2.4.

5.2.1 Extraction and families

In this section, we shall consider standard derivations only.

Proposition 5.2.7 *If there exists a reduction* τT *such that* $\rho R \trianglerighteq \tau T \trianglelefteq \sigma S$, *then* $\rho R \simeq \sigma S$.

Proof It is enough to observe that, if $\rho R \trianglerighteq \tau T$, then $\rho R \simeq \tau T$. In the first three cases in the definition of $\rhd$, we obviously have $\tau T \leq \rho R$. In the last case (see Figure 5.3) we have $\rho = \rho' R' \rho''$ and $\tau = \rho' \tau'$, for some R'. Let $\nu = \tau'/(R'\rho'')$. Then $\rho''\nu \equiv \tau'/R'$. Moreover, R has a unique residual T' after ν, and this must also be a residual of T after R'/τ'. (Note that T is internal to N' in $C[(\lambda.\, x\ N')]$, and that R is the image of T in $C[M^x[N, \ldots, N', \ldots, N]]$.) Thus, $\rho R \leq \rho'(R' \sqcup \tau')T' \geq \tau T$, that implies $\rho R \simeq \tau T$. □

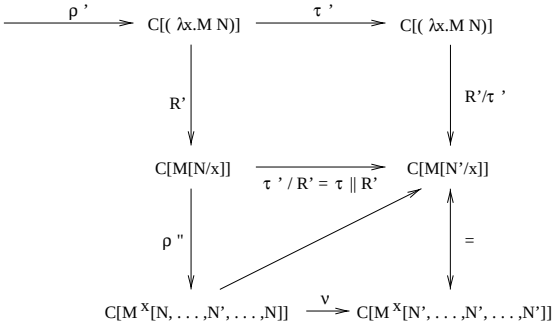


Fig. 5.3.

Also the converse of the previous proposition is true. But to prove it we need some preliminary lemmas.

Lemma 5.2.8 *If* $R \leq \sigma S$, *then* $\sigma S \trianglerighteq R$.

Proof If $R \leq \sigma S$, then $S \in R/\sigma$ and $\sigma S \trianglerighteq R$ by $\rhd_1$. □

Lemma 5.2.9 *Let* $\rho R \leq \sigma S$. *There is a reduction* τT *such that* $\rho R \trianglerighteq \tau T \trianglelefteq \sigma S$.

Proof First of all, let us note that $\rho \sqsubseteq \sigma$, for by hypothesis $\rho R \leq \sigma S$. The proof continues then by induction on $|\sigma|$. The base case is immediate. In fact, when $|\sigma| = 0$, also $|\rho| = 0$, for $\rho \sqsubseteq \sigma$. The result follows then by Lemma 5.2.8. So, let us proceed with the induction case.
We can distinguish two subcases, according to the length of ρ. The

easy one is $|\rho| = 0$, for the result follows again by Lemma 5.2.8. Thus, let $\sigma = S'\sigma'$ and $\rho = R'\rho'$. If $S' = R'$, then $\rho'R \leq \sigma'S$ and, by the induction hypothesis, there exists $\tau'T$ such that $\rho'R \trianglerighteq \tau'T \trianglelefteq \sigma'S$. Then, for $\tau = R'\tau'$, $\rho R \trianglerighteq \tau T \trianglelefteq \sigma S$.

Summarizing, we have left to prove the case $\sigma = S'\sigma'$, $\rho = R'\rho'$, and $R' \neq S'$. The redex R' cannot be external or to the left of S', for otherwise $R'/\sigma' \neq \emptyset$, contradicting the hypothesis $\rho \sqsubseteq \sigma$. Since ρ is standard (recall the assumption at the beginning of this section), it can be decomposed as $\rho = \rho_f \sqcup \rho_a \sqcup \rho_d$ such that ρ_f, ρ_a and ρ_d are respectively internal to the function part of S', internal to the argument part of S', and disjoint from S' (to its right). Moreover, S' has a *unique* residual S'' after ρ (remind that S' is leftmost w.r.t. Rq'). We proceed then by case analysis, according to the mutual positions of R and S'' in the final term of ρ.

(i) R is external or to the left of S'': There is a redex T external or to the left of S' such that $R \in T/\rho$ (easy induction on $|\sigma|$). Thus, $\rho R \trianglerighteq T$. That, by Proposition 5.2.7, implies $T \simeq \rho R$. Moreover, $T \simeq \sigma S$, for $\rho R \leq \sigma S$. Hence, $S \in T/\sigma$ (see Exercise 5.1.14) and $\sigma S \trianglerighteq T$ (by definition of $\rhd_1$).

(ii) R is internal to the function part of S'': Let $\nu = (\rho_a \sqcup \rho_d)/\rho_f$. If S_f is the unique residual of S' after ρ_f, then $S_f/\nu = \{S''\}$. Moreover, ν is disjoint from the function part of S_f. Thus, there exists a redex R_f in the function part of S_f such that $R_f/\nu = \{R\}$, that implies both $\rho R \trianglerighteq \rho_f R_f$ and $\rho R \geq \rho_f R_f$. By transitivity, since $\rho R \leq \sigma S$, we also have $\rho_f R_f \leq \sigma S$. By Lemma 5.1.11, $S \in R_f/(\sigma/\rho_f)$. Then, let $\rho'_f = \rho_f/S'$ and $R'_f = R_f/S_f$ (i.e., $(\rho_f R_f)/S' = \rho'_f R'_f$). We have that $\rho'_f R'_f \leq \sigma' S$ and $S'\rho'_f R'_f \trianglerighteq \rho_f R_f$ (see Figure 5.4). Since ρ_f is in the function part of S' and ρ_f

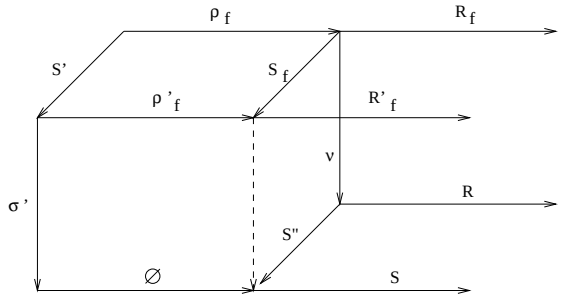


Fig. 5.4.

is standard, $S'\rho'_f$ and then ρ'_f are standard too. We can now apply the induction hypothesis to $\rho'_f R'_f \leq \sigma' S$, concluding that there is $\tau' T'$ such that $\rho'_f R'_f \trianglerighteq \tau' T' \trianglelefteq \sigma' S$. Therefore, $S'\rho'_f R'_f \trianglerighteq S'\tau' T' \trianglelefteq \sigma S$ (let us note that also τ' is in the function part of S', and then that $S'\tau'$ too is standard). But we have already seen that $S'\rho'_f R'_f \trianglerighteq \rho_f R_f$. So, by the Church-Rosser property of $\trianglerighteq$, there exists a derivation τT such that $S'\tau' T' \trianglerighteq \tau T \trianglelefteq \rho_f R_f$. Thus, $\sigma S \trianglerighteq \tau T \trianglelefteq \rho R$.

(iii) R is disjoint from S'' or to its right: There is again some R_d disjoint from the residual S_d of S' after ρ_d, such that $\rho R \trianglerighteq \rho_d R_d$ and $\rho R \geq \rho_d R_d$. Then we proceed as in the previous case.

(iv) R is in the argument part of S'': There is a redex R_a in the argument part of the residual S_a of S' after ρ_a, such that $\rho R \trianglerighteq \rho_a R_a$ and $\rho R \geq \rho_a R_a$. Unfortunately, the previous reasoning does not go through so simply in this case, since $\rho''_a = \rho_a / S'$ might not be standard. In fact, let us note that ρ''_a is the union of disjoint reductions, each of them internal to a different instance of the argument of S_a in its contractum. Anyhow, it is still true that $\rho_a R_a \leq \sigma S$, for $\rho R \leq \sigma S$. Thus, $S \in R_a/(\sigma/\rho_a)$ and there is a redex $R''_a \in R_a/S_a$ such that $\rho''_a R''_a \leq \sigma' S$. Moreover, R''_a is internal to some instance, say the i-th one, of the argument part of S_a in its contractum. Let us take the component ρ'_a of ρ''_a internal to such an instance. There is a redex R'_a such that $\rho'_a R'_a \leq \rho R$, with $\rho'_a R'_a$ standard. Moreover, $S'\rho'_a R'_a \trianglerighteq \rho_a R_a$, for $\rho_a R_a / S' = (\rho'_a R'_a) \| S'$ by $\rhd_4$; and $\rho'_a R'_a \leq \sigma' S$, for $\rho''_a R''_a \leq \sigma' S$. We can now proceed as in case 2. By induction hypothesis, there is $\tau' T'$ such that $\rho'_a R'_a \trianglerighteq \tau' T' \trianglelefteq \sigma' S$. Therefore, $S'\rho'_a R'_a \trianglerighteq S'\tau' T' \trianglelefteq \sigma S$. But since $S'\rho'_a R'_a \trianglerighteq \rho_a R_a$, by the Church-Rosser property of $\trianglerighteq$, there exists τT such that $S'\tau' T' \trianglerighteq \tau T \trianglelefteq \rho_a R_a$. Thus, $\sigma S \trianglerighteq \tau T \trianglelefteq \rho R$.

□

Let us remark the structure of the previous proof. It follows exactly the definition of $\trianglerighteq$. In fact, each subcase in Definition 5.2.2 yields a corresponding subcase in the non-trivial part of the proof. It is indeed true that this is the actual reason of the four cases in the definition of extraction.

Proposition 5.2.10 *If* $\rho R \simeq \sigma S$, *then* $\rho R \trianglerighteq \tau T \trianglelefteq \sigma S$ *for some* τT.

Proof By definition of family, $\rho R \simeq \sigma S$ if and only if there exists a chain of $\rho_i R_i$ such that $\rho_0 R_0 = \rho R$, $\rho_n R_n = \sigma S$, and for all $1 \leq i \leq n$ either $\rho_{i-1} R_{i-1} \leq \rho_i R_i$ or $\rho_i R_i \leq \rho_{i-1} R_{i-1}$ (remind that we can assume without loss of generality that all the ρ_i are standard). By Lemma 5.2.9, there exists $\tau_i T_i$ such that $\rho_{i-1} R_{i-1} \trianglerighteq \tau_i T_i \trianglelefteq \rho_i R_i$, for every i. By the confluence of $\trianglerighteq$, we conclude then that there exists τT such that $\rho R \trianglerighteq \tau T \trianglelefteq \sigma S$. □

Theorem 5.2.11 (decidability of extraction) *Let* σ *and* ρ *be two* standard *reductions. Then* $\rho R \simeq \sigma S$ *if and only if* $\rho R \trianglerighteq \tau T \trianglelefteq \sigma S$ *for some* τT*.*

Proof By Proposition 5.2.7 and Proposition 5.2.10. □

The previous result not only establishes a full correspondence between zig-zag and extraction, but also gives an *effective* procedure for deciding the family relation. In fact, given two redexes ρR and σS, there is an effective way for deriving the standard reductions ρ_s and σ_s respectively equivalent to ρ and σ by permutation. Then, as extraction is effective and terminating, we can compute the $\trianglerighteq$ canonical forms $\rho'_s R'$ and $\sigma'_s S'$ of $\rho_s R$ and $\sigma_S S$. If and only if $\rho'_s R' = \sigma'_s S'$ the redexes ρR and σS are in the same family. The previous considerations can be summarized as follows.

Definition 5.2.12 (canonical derivation) Every standard derivation σu in normal form with respect to $\trianglerighteq$ will be called *canonical.*

Corollary 5.2.13 (canonical representative) *The* canonical representative *of a family can be effectively derived from each member* ρR *of the family: it is the unique canonical derivation* $\rho_c R_c$ *such that* $\rho_s R \trianglerighteq \rho_c R_c$*, where* ρ_s *is the standard derivation equivalent to* ρ*. Each family of redexes has a unique canonical representative.*

5.3 Labeling

The *labeled* λ*-calculus* is an extension of λ-calculus proposed by Lévy in [Lév78].

Let $L = \{a, b, \cdots\}$ be a denumerable set of *atomic labels.* The set $\mathbf{L}$ of *labels*, ranged over by $\alpha, \beta, \cdots$, is defined as the set of words over the alphabet L, with an arbitrary level of nested underlinings and overlinings.

Formally, $\mathbf{L}$ is the smallest set containing L and closed with respect to the following formation rules:

(i) if $\alpha \in \mathbf{L}$ and $\beta \in \mathbf{L}$, then $\alpha\beta \in \mathbf{L}$;
(ii) if $\alpha \in \mathbf{L}$, then $\underline{\alpha} \in \mathbf{L}$;
(iii) if $\alpha \in \mathbf{L}$, then $\overline{\alpha} \in \mathbf{L}$.

The operation of concatenation $\alpha\beta$ is supposed to be associative.

The set $\Lambda_V^{\mathbf{L}}$ of labeled λ-terms over a set V of variables and a set $\mathbf{L}$ of labels is defined as the smallest set containing:

(i) x^α, for any $x \in V$ and $\alpha \in \mathbf{L}$;
(ii) $(M\ N)^\alpha$, for all $M, N \in \Lambda_V^{\mathbf{L}}$ and $\alpha \in \mathbf{L}$;
(iii) $(\lambda x.M)^\alpha$, for any $M \in \Lambda_V^{\mathbf{L}}$ and $\alpha \in \mathbf{L}$.

As usual, we shall identify terms up to α-conversion.

The concatenation $\alpha \cdot M$ of a label α with a labeled term M is defined as follows:

(i) $\alpha \cdot x^\beta = x^{\alpha\beta}$
(ii) $\alpha \cdot (M\ N)^\beta = (M\ N)^{\alpha\beta}$
(iii) $\alpha \cdot (\lambda x.M)^\beta = (\lambda x.M)^{\alpha\beta}$

The substitution $M[N/x]$ of a free variable x for a labeled λ-term N in a labeled λ-term M, is inductively defined by:

(i) $x^\alpha[N/x] = \alpha \cdot N$
(ii) $y^\alpha[N/x] = y^\alpha$, when $y \neq x$
(iii) $(M_1\ M_2)^\alpha[N/x] = (M_1[N/x]\ M_2[N/x])^\alpha$
(iv) $(\lambda x.\, M)^\alpha[N/x] = (\lambda x.\, M)^\alpha$
(v) $(\lambda y.\, M)^\alpha[N/x] = (\lambda y.\, M[N/x])^\alpha$

In item 5 above, N is free for y in M, that is, no free variable of N is captured by the binder of y. This is not a limitation, due to our assumption on α-conversion, we can always suitably rename the variable y in $\lambda y.\, M$.

In the labeled system, β-reduction is defined by the following rule:

$$((\lambda x.M)^\alpha\ N)^\beta \to \beta \cdot \overline{\alpha} \cdot M[\underline{\alpha} \cdot N/x]$$

The *degree* of a redex $R = ((\lambda x.M)^\alpha\ N)^\beta$ is the label α of its function part (notation: $\mathsf{degree}(R). = \alpha$).

The formal presentation of the labeled λ-calculus given above should not scare the reader. The main idea is indeed very simple, once shifting from the concrete syntax of terms to their graphical representation as

syntax trees. In this context, to add a label to each subterm corresponds to mark each edge in the tree by a label. The degree of a redex $R = ((\lambda x.M)^{\alpha}\ N)^{\beta}$ is the label of the edge between the corresponding @-node of $(\lambda x.M\ N$ and the λ-node of $\lambda x.M$. Firing the redex, the degree of R is captured between the labels of the other edges incident to the the nodes in R. Namely, between the label of the application of R and the label of the body of the abstraction in R, and between the label of any occurrence of the variable in R and the label of the argument part of R. In the contractum, the pairs of edges corresponding to the previous pairs of labels are replaced by new connections. Apart from lining, the labels of these new edges are obtained composing the labels of the corresponding edges in the natural way (see Figure 5.5).

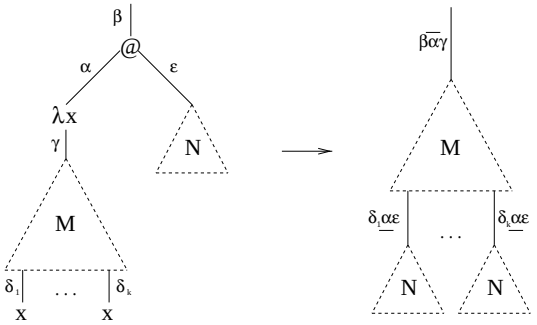


Fig. 5.5. Labeled β-reduction

Overlining and underlining respectively represent the two ways in which new connections are created firing R: upwards (from the context to the body M of $\lambda x.M$), and downwards (from the occurrences of the variable x in M to the instances of the argument N).

Example 5.3.1 We already observed that, contracting (I (I x)), after one step we get the same term (I x) whichever redex we reduced. Besides, we pointed out that this equivalence of the result should have been considered merely incidental—for our purposes, a sort of lack in the syntax of λ-calculus. This situation is correctly handled by the labeled system, where labels allow to distinguish between the way in which the two terms (I x) are obtained (see Figure 5.6). Besides, let us also note that, after one more step, the reduction ends with the same term also in the labeled system (Figure 5.6). In fact, labeling preserves confluence of the calculus, as we will show in the next section.

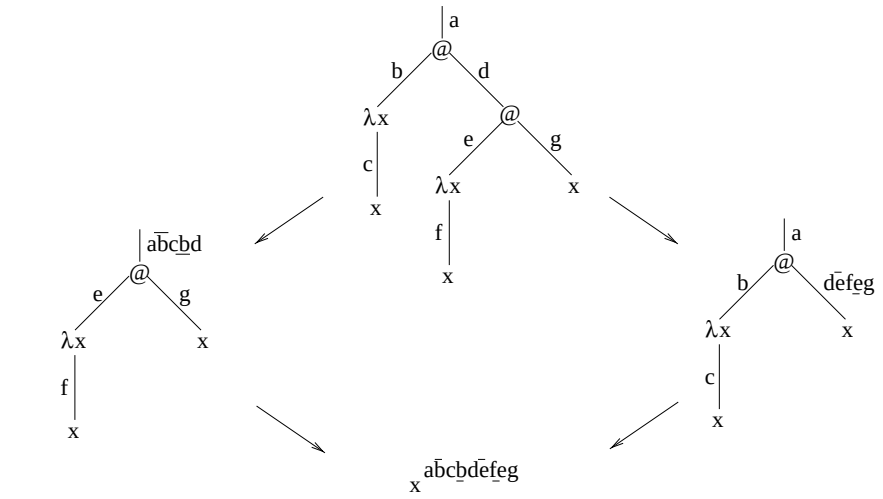


Fig. 5.6. (I (I x))

Example 5.3.2 In the λ-calculus, reducing $(\Delta\ \Delta)$ we get a reduction sequence composed of an infinite number of copies of $(\Delta\ \Delta)$. As for the previous example, in the labeled calculus this is not true anymore. In fact, the reduction of $(\Delta\ \Delta)$ gives rise to an infinite sequence of distinguished labeled terms (see Figure 5.7).

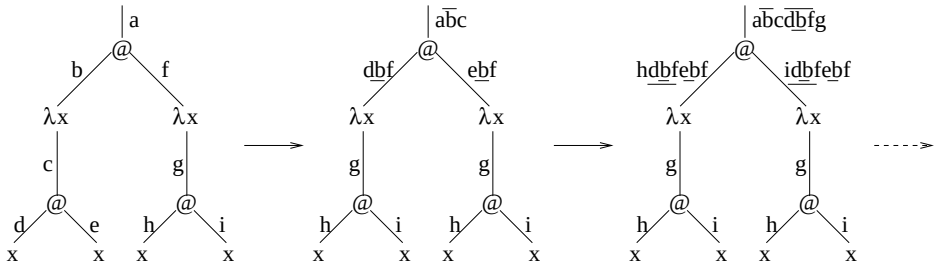


Fig. 5.7. (λx.(x x) λx.(x x))

5.3.1 Confluence and Standardization

In this section we shall prove the Church-Rosser and standardization properties for the labeled λ-calculus.

For the sake of the proof of such properties, we will consider a suitable extension of β-reduction. Namely, we will assume that the degrees of the redex contracted reducing terms verify a given predicate $\mathcal{P}$ on $\mathbf{L}$. According to this, the reduction $((\lambda x.M)^{\alpha}\, N)^{\beta} \to \beta \cdot \overline{\alpha} \cdot M[\underline{\alpha} \cdot N/x]$ will be considered legal only when $\mathcal{P}(\alpha)$ is true.

By the way, the introduction of this predicate does not cause any loss of expressiveness, as usual β-reduction corresponds to the case in which $\mathcal{P}$ is always true. Furthermore, the use of $\mathcal{P}$ allows to recover in a very simple way many other kinds of labelings considered in the literature, without a sensible complication of the theory of the calculus.

The schema of the proofs of confluence and standardization is the following:

(i) We shall start proving that labeled λ-calculus is locally confluent, for any choice of the predicate $\mathcal{P}$.

(ii) We shall directly prove standardization for strongly normalizable terms.

(iii) Since it is well known that local confluence implies confluence for strongly normalizable terms, the goal will be to exploit the previous results in conjunction with some suitable predicate $\mathcal{P}$ ensuring strong normalization. To this purpose, we shall give some sufficient conditions for $\mathcal{P}$ that imply strong normalization of every term (taking into account reductions that are legal with respect to $\mathcal{P}$ only).

(iv) Finally, we shall observe that, for any pair of reductions $\rho : M \twoheadrightarrow N$ and $\rho' : M \twoheadrightarrow N'$, we can construct a predicate $\mathcal{P}$ satisfying the above mentioned sufficient conditions, for which the reductions ρ and ρ' are legal.

The proof of local confluence is preceded by some lemma useful to prove that substitution behaves well with respect to labels. Namely, that the usual property of λ-calculus $M[N/x] \twoheadrightarrow M'[N'/x]$, when $M \twoheadrightarrow M'$ and $N \twoheadrightarrow N'$, holds in the labeled case too.

Lemma 5.3.3

(i) $\alpha \cdot (M[N/x]) = (\alpha \cdot M)[N/x]$

(ii) $M[N/x][N'/y] = M[N'/x][N[N'/y]/x]$, *when* $x \neq y$ *and* x *does not occur free in* N'.

Proof An easy induction on the structure of M. □

Lemma 5.3.4 *If* $M \twoheadrightarrow M'$, *then* $M[N/x] \twoheadrightarrow M'[N/x]$.

Proof Let us prove first the case $M \to M'$ by structural induction on M:

(i) $M = x^\alpha$. This case is vacuous.

(ii) $M = (\lambda x.M_1)^\alpha$. Since the redex must be internal to M_1, $M' = (\lambda x.M_1')^\alpha$, with $M_1 \to M_1'$. By induction hypothesis, $M_1[N/x] \to M_1'[N/x]$, and thus $M[N/x] \to M'[N/x]$.

(iii) $M = (M_1\ M_2)^\alpha$, and the redex is internal to M_1 or M_2. This case is similar to the previous one.

(iv) $M = ((\lambda y.M_1)^\alpha\ M_2)^\beta$, $M' = \beta\overline{\alpha} \cdot M_1[\underline{\alpha} \cdot M_2/y]$ and $\mathcal{P}(\alpha)$ is true. By α-conversion we may suppose $x \neq y$ and y not free in N. Then, we have $M[N/x] = ((\lambda y.M_1[N/x])^\alpha\ M_2[N/x])^\beta$, and

$$\begin{array}{lcll} M'[N/x] & = & (\beta\overline{\alpha} \cdot M_1[\underline{\alpha} \cdot M_2/y])[N/x] & \\ & = & \beta\overline{\alpha} \cdot M_1[\underline{\alpha} \cdot M_2/y][N/x] & \text{by Lemma 5.3.3.1} \\ & = & \beta\overline{\alpha} \cdot M_1[N/x][(\underline{\alpha} \cdot M_2)[N/x]/y] & \text{by Lemma 5.3.3.2} \\ & = & \beta\overline{\alpha} \cdot M_1[N/x][\underline{\alpha} \cdot M_2[N/x]/y] & \text{by Lemma 5.3.3.1} \end{array}$$

Since $\mathcal{P}(\alpha)$ is true, $M[N/x] \to M'[N/x]$.

By iteration, we get the case $M \twoheadrightarrow M'$. □

Lemma 5.3.5 *If* $M \twoheadrightarrow M'$, *then* $\alpha \cdot M \twoheadrightarrow \alpha \cdot M'$.

Proof Trivial. □

Lemma 5.3.6 *If* $N \twoheadrightarrow N'$, *then* $M[N/x] \twoheadrightarrow M[N'/x]$.

Proof By structural induction on M:

(i) $M = x^\alpha$. Then, $M[N/x] = \alpha \cdot N$, $M[N'/x] = \alpha \cdot N'$, and $\alpha \cdot N \twoheadrightarrow \alpha \cdot N'$, by Lemma 5.3.5.

(ii) $M = y^\alpha$, where $y \neq x$. Obvious.

(iii) $M = (\lambda x.M_1)^\alpha$. Then (up to α-conversion), $M[N/x] = (\lambda x.M_1[N/x])^\alpha$, and $M[N'/x] = (\lambda x.M_1[N'/x])^\alpha$. By induction hypothesis, $M_1[N/x] \twoheadrightarrow M_1[N'/x]$, and thus $M[N/x] \twoheadrightarrow M[N'/x]$.

(iv) $M = (M_1\ M_2)^\alpha$. Analogous to the previous case.

□

Corollary 5.3.7 *If* $M \twoheadrightarrow M'$ *and* $N \twoheadrightarrow N'$ *then,* $M[N/x] \twoheadrightarrow M'[N'/x]$.

Proof By Lemma 5.3.4 and Lemma 5.3.6. □

Proposition 5.3.8 (local confluence) *For any pair of redexes* $M \xrightarrow{R} M'$ *and* $M \xrightarrow{S} M''$*, there exist* N *and two reductions* ρ *and* σ *such that* $M' \xrightarrow{\sigma} N$ *and* $M'' \xrightarrow{\rho} N$*.*

Proof By structural induction on M:

(i) $M = x^{\alpha}$. This case is vacuous.

(ii) $M = (\lambda x.M_1)^{\alpha}$. Since R and S must be internal to M_1, $M' = (\lambda x.M_1')^{\alpha}$, $M'' = (\lambda x.M_1'')^{\alpha}$, $M_1 \to M_1'$, and $M_1 \to M_1''$. By induction hypothesis, there exists N_1 such that $M_1' \twoheadrightarrow N_1$ and $M_1'' \twoheadrightarrow N_1$. Taking $N = (\lambda x.N_1)^{\alpha}$ we have done.

(iii) $M = (M_1\ M_2)^{\alpha}$, and both redexes R and S are either internal to M_1 or internal to M_2. This case is similar to the previous one.

(iv) $M = (M_1\ M_2)^{\alpha}$, R is internal to M_1 and S is internal to M_2 (or vice versa). In this case, we close the diamond in one step, firing the unique residuals of R and S in M'' and M', respectively..

(v) $M = ((\lambda y.M_1)^{\alpha}\ M_2)^{\beta} \xrightarrow{R} M' = \beta\overline{\alpha} \cdot M_1[\underline{\alpha} \cdot M_2/y]$. We distinguish two subcases:

  (a) S is internal to M_1. Let M_1' be the reduct of M_1 by S. Then, $M'' = ((\lambda y.M_1')^{\alpha}\ M_2)^{\beta} \to N = \beta\overline{\alpha} \cdot M_1'[\underline{\alpha} \cdot M_2/y]$. Since $M_1 \to M_1'$, by Lemma 5.3.4 and Lemma 5.3.5, $M' = \beta\overline{\alpha} \cdot M_1[\underline{\alpha} \cdot M_2/y] \to N = \beta\overline{\alpha} \cdot M_1'[\underline{\alpha} \cdot M_2/y]$.

  (b) S is internal to M_2. Let M_2' be the reduct of M_2 by S. Then, $M'' = ((\lambda y.M_1)^{\alpha}\ M_2')^{\beta} \to N = \beta\overline{\alpha} \cdot M_1[\underline{\alpha} \cdot M_2'/y]$. Since $M_2 \to M_2'$, by Lemma 5.3.5 and Lemma 5.3.6, $M' = \beta\overline{\alpha} \cdot M_1[\underline{\alpha} \cdot M_2/y] \twoheadrightarrow N = \beta\overline{\alpha} \cdot M_1[\underline{\alpha} \cdot M_2'/y]$.

(vi) As in the previous case, inverting R and S.

It is immediate to check that, extending the definitions of residual and development to the labeled case (see Definition 5.1.2 and Theorem 5.1.3), we have indeed that $\rho = S/R$ and $\sigma = S/R$. □

Remark 5.3.9 Let us note again that the ρ and σ are legal for any predicate $\mathcal{P}$ such that both $\mathcal{P}(R)$ and $\mathcal{P}(S)$ are true. Indeed, since $\rho = S/R$ and $\sigma = S/R$, the degree of all the redexes fired in ρ (σ) is equal to $\mathsf{degree}\,S$ ($\mathsf{degree}\,R$). This might not be surprising, since labels aim at mark redexes with the same origin, closing the diamond of local confluence we should use instances of the redex on the opposite side of the diamond.

This is indeed the idea that we will pursue finding a suitable predicate $\mathcal{P}$ for any pair of reductions starting from the same term (see the proof of Theorem 5.3.26).

It is a well known result of rewriting systems that local confluence implies confluence for strongly normalizable terms (in literature this property is known as Newman Lemma). The proof is a simple induction on the length of the longest normalizing derivation for the term. So, we have the following corollary.

Proposition 5.3.10 (confluence) *Let* M *be a strongly normalizable term. If* $\rho : M \twoheadrightarrow M'$ *and* $\sigma : M \twoheadrightarrow M''$*, then there exist* N *such that* $\rho' : M' \twoheadrightarrow N$ *and* $\sigma' : M'' \twoheadrightarrow N$*, where* $\rho' = \sigma/\rho$ *and* $\sigma' = \rho/\sigma$*.*

Proof By Newman's Lemma and Lemma 5.3.8 we have confluence, that is, the existence of N, $\rho' : M' \twoheadrightarrow N$ and $\sigma' : M'' \twoheadrightarrow N$. Anyhow, since we want to prove that $\rho' = \sigma/\rho$ and $\sigma' = \rho/\sigma$, let us give the proof in full details. Let us define a measure $\mathsf{depth}(\,.\,)$ on terms, such that $\mathsf{depth}(T)$ is the length of the longest reduction of T. The proof is by induction on $\mathsf{depth}(M)$.

(i) $\mathsf{depth}(M) = 0$. M is in normal form. Hence, $\rho = \sigma = \epsilon$ and $\rho' = \sigma' = \epsilon$

(ii) The cases $\rho = \epsilon$ or $\sigma = \epsilon$ are trivial, since we immediately see that $\rho' = \sigma$ and $\sigma' = \rho$. Hence, let us assume $\rho = M' \xrightarrow{R} P' \overset{\rho_0}{\twoheadrightarrow} M'$ and $\rho = M' \xrightarrow{S} P'' \overset{\sigma_0}{\twoheadrightarrow} M''$. By Lemma 5.3.8, there exists Q such that $S/R : T' \twoheadrightarrow Q$ and $R/S : T'' \twoheadrightarrow Q$. By definition, $\mathsf{depth}(T'), \mathsf{depth}(T'') < \mathsf{depth}(M)$. Hence, by induction hypothesis, there are Q' and Q'' such that $\rho_0/(S/R) : Q \twoheadrightarrow Q'$, $\sigma_0/(R/S) : Q \twoheadrightarrow M'$, $(S/R)/\rho_0 : M' \twoheadrightarrow Q'$ and $(R/S)/\sigma_0 : M'' \twoheadrightarrow Q''$. Finally, since $\mathsf{depth}(Q) \leq \mathsf{depth}(T')$ and $\mathsf{depth}(Q) \leq \mathsf{depth}(T'')$, by induction hypothesis we have that $(\sigma_0/(R/S))/(\rho_0/(S/R)) : Q' \twoheadrightarrow N$ and $(\rho_0/(S/R))/(\sigma_0/(R/S)) : Q'' \twoheadrightarrow N$, for some N. It is now an easy exercise to verify that $R\rho_0((S/R)/\rho_0)((\sigma_0/(R/S))/(\rho_0/(S/R))) = \sigma/\rho$ and that $S\sigma_0((R/S)/\sigma_0)((\rho_0/(S/R))/(\sigma_0/(R/S))) = \sigma/\rho$.

□

Exercise 5.3.11 Verify the last two equivalences in the proof of Theorem 5.3.10. Moreover, we invite the reader to draw the reduction diagram corresponding to the previous proof.

Exercise 5.3.12 We invite the reader to reflect on why the strong normalization hypothesis is mandatory in Newman Lemma. Then, we invite him to prove that the following rewriting system

$$\mathtt{a} \to \mathtt{b} \qquad \mathtt{b} \to \mathtt{a} \qquad \mathtt{a} \to \mathtt{c} \qquad \mathtt{b} \to \mathtt{d}$$

(over the set of symbolsa $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$) is not confluent, although it is locally confluent.

Let us now prove standardization for strongly normalizable terms. The definition of standard and normal (leftmost-outermost) derivations are as usual:

- A reduction $M_0 \xrightarrow{R_1} M_1 \xrightarrow{R_2} \cdots \xrightarrow{R_k} M_k \xrightarrow{R_{k+1}} \cdots$ is *standard* when for any i, j such that $1 \leq i \leq j$, the redex R_j is not residual of a redex in M_{i-1} to the left of R_i.
- A reduction $M_0 \xrightarrow{R_1} M_1 \xrightarrow{R_2} \cdots \xrightarrow{R_k} M_k \xrightarrow{R_{k+1}} \cdots$ is *normal* (or leftmost-outermost) when for any $i \geq 1$ the redex R_i is the leftmost redex in M_i.

In the following, we shall respectively use $M \overset{st}{\twoheadrightarrow} N$ and $M \overset{norm}{\twoheadrightarrow} N$ to denote standard and normal reductions.

It is immediate that any normal reduction is also standard and that normal reduction is uniquely determined. Namely, if $\rho : M \overset{norm}{\twoheadrightarrow} N$ and $\rho' : M \overset{norm}{\twoheadrightarrow} N'$, then ρ is a prefix of ρ', or vice versa.

Definition 5.3.13 The function Abs is defined as follows:

(i) $\mathsf{Abs}((\lambda x.M)^\alpha) = (\lambda x.M)^\alpha$

(ii) $\mathsf{Abs}(M\ N)^\alpha) = \mathsf{Abs}(\alpha\overline{\beta} \cdot P[\underline{\beta} \cdot N/x])$ if $\mathsf{Abs}(M) = (\lambda x.P)^\alpha$

So, $\mathsf{Abs}(M)$ is the first abstraction obtained by normal reduction of M. Obviously, Abs is not always defined.

Example 5.3.14 Variables are trivial examples of terms for which Abs is undefined. Nevertheless, for some terms the reason for which Abs is undefined are more deep. For instance, let us take again $(\Delta\,\Delta)$. Since, $(\Delta\,\Delta) \to (\Delta\,\Delta)$ is the only contraction of $(\Delta\,\Delta)$, we can never obtain an abstraction along its reduction.

Lemma 5.3.15 *Every standard reduction* $M \overset{st}{\twoheadrightarrow} (\lambda x.N)^\alpha$ *can be decomposed in the following way:* $M \overset{norm}{\twoheadrightarrow} \mathsf{Abs}(M) \twoheadrightarrow (\lambda x.N)^\alpha$

Proof Trivial. □

Proposition 5.3.16 *Let* M *be strongly normalizable. For any* $\rho : M \twoheadrightarrow N$, *there exists a corresponding standard reduction* $\sigma : M \overset{st}{\twoheadrightarrow} N$, *such that* $\rho \equiv \sigma$.

Proof Let us call $\mathsf{depth}(M)$ the length of the longest normalizing derivation for M. The proof is by double induction over $\mathsf{depth}(M)$ and the structure of M.
If $\mathsf{depth}(M) = 0$ the result is trivial. If $\mathsf{depth}(M) > 0$, we proceed instead by structural induction on M:

(i) $M = x^{\alpha}$. This case is vacuous.

(ii) $M = (\lambda x.M_1)^{\alpha}$. Then, $N = (\lambda x.N_1)^{\alpha}$, and $M_1 \twoheadrightarrow N_1$. Furthermore, M_1 is a subterm of M and $\mathsf{depth}(M_1) = \mathsf{depth}(M)$. So, by induction hypothesis, $M_1 \overset{st}{\twoheadrightarrow} N_1$, and $M \overset{st}{\twoheadrightarrow} N$.

(iii) $M = (M_1\ M_2)^{\alpha}$. Let us distinguish two subcases:

(a) $N = (N_1\ N_2)^{\alpha}$, and the reduction $M \twoheadrightarrow N$ is composed of two separate reductions internal to M_1 and M_2. Then, by induction hypothesis, $M_1 \overset{st}{\twoheadrightarrow} N_1$, $M_2 \overset{st}{\twoheadrightarrow} N_2$, and $M = (M_1\ M_2)^{\alpha} \overset{st}{\twoheadrightarrow} (N_1\ M_2)^{\alpha} \overset{st}{\twoheadrightarrow} (N_1\ N_2)^{\alpha} = N$.

(b) The reduction $M \twoheadrightarrow N$ can be decomposed in the following way: $M = (M_1\ M_2)^{\alpha} \twoheadrightarrow ((\lambda x.N_1)^{\beta}\ M_2)^{\alpha} \to \alpha\overline{\beta} \cdot N_1[\underline{\beta} \cdot N_2/x] \twoheadrightarrow N$, where $M_2 \twoheadrightarrow N_2$ and $M_1 \twoheadrightarrow (\lambda x.N_1)^{\beta}$. By induction hypothesis, $M_1 \overset{st}{\twoheadrightarrow} (\lambda x.N_1)^{\beta}$, and by Lemma 5.3.15, $M_1 \overset{norm}{\twoheadrightarrow} (\lambda x.M_3) \overset{st}{\twoheadrightarrow} (\lambda x.N_1)^{\beta}$, where $\mathsf{Abs}(M_1) = (\lambda x.M_3)^{\beta}$. So, $M_3 \twoheadrightarrow N_1$. Moreover, $M_2 \twoheadrightarrow N_2$, and by Lemma 5.3.5 and Lemma 5.3.6, $\alpha\overline{\beta} \cdot M_3[\underline{\beta} \cdot M_2/x] \twoheadrightarrow \alpha\overline{\beta} \cdot N_1[\underline{\beta} \cdot N_2/x]$. Summing up,

$$\begin{aligned} M = (M_1\ M_2)^{\alpha} &\overset{norm}{\twoheadrightarrow} ((\lambda x.M_3)^{\beta}\ M_2)^{\alpha} \\ &\to \alpha\overline{\beta} \cdot M_3[\underline{\beta} \cdot M_2/x] \twoheadrightarrow \alpha\overline{\beta} \cdot N_1[\underline{\beta} \cdot N_2/x] \twoheadrightarrow N \end{aligned}$$

Let us now note that $\mathsf{depth}(\alpha\overline{\beta} \cdot M_3[\underline{\beta} \cdot M_2/x]) < \mathsf{depth}(M)$. So, by induction hypothesis there exists $\alpha\overline{\beta} \cdot M_3[\underline{\beta} \cdot M_2/x] \overset{st}{\twoheadrightarrow} N$. So, we finally have the standard reduction:

$$M = (\lambda x.M_3)^{\beta}\ M_2)^{\alpha} \overset{norm}{\twoheadrightarrow} ((\lambda x.N_1)^{\beta}\ M_2)^{\alpha}$$

We leave as an exercise to the reader to verify that the reduction σ built in the proof and ρ are equivalent by permutation. □

Remark 5.3.17 Proposition 5.3.10 and Proposition 5.3.16 hold for *every* choice of the predicate $\mathcal{P}$. In particular, assuming the usual notion of β-reduction, they hold for any subset of λ-terms that is strongly normalizable—for instance, strong normalization might be obtained restricting our attention to terms typable according to a suitable typing discipline. Anyhow, we want to prove confluence and standardization for *any* λ-term. So, the strategy must be to restrict β-reduction, eliminating those reductions that cause non-termination. At the same time, we want that the reductions involved in the theorem remain legal. Hence, next step is to find some sufficient conditions on $\mathcal{P}$ such that all terms of labeled λ-calculus become strongly normalizable.

Let us start introducing some notation and definitions.

Notation 5.3.18 We shall use $\ell(M)$ to denote the external label of a labeled λ-term. In particular, $\ell(x^\alpha) = \ell((M\ N)^\alpha) = \ell((\lambda x.M)^\alpha) = \alpha$.

Definition 5.3.19 The height $h(\alpha)$ of a label α is the maximal nesting depth of overlinings and underlinings in it. Formally:

(i) $h(a) = 0$, when a is an atomic label;
(ii) $h(\alpha\beta) = \max\{h(\alpha), h(\beta)\}$;
(iii) $h(\overline{\alpha}) = h(\underline{\alpha}) = 1 + h(\alpha)$.

Definition 5.3.20 We say that the predicate $\mathcal{P}$ *has an upper bound*, if the set $\{h(\alpha) \mid \mathcal{P}(\alpha)\}$ has an upper bound.

Lemma 5.3.21 *If* $M \twoheadrightarrow M'$, *then* $h(\ell(M)) \leq h(\ell(M'))$.

Proof Obvious. □

Lemma 5.3.22 *Let* $T = (\dots((M\ N_1)^{\beta_1}\ N_2)^{\beta_2} \dots N_n)^{\beta_n}$. *If* $T \twoheadrightarrow (\lambda x.N)^\alpha$, *then* $h(\ell(M)) \leq h(\alpha)$.

Proof By induction on n. When $n = 0$, the result follows by the previous lemma. When $n > 0$, we must have:

- $(\dots((M\ N_1)^{\beta_1}\ N_2)^{\beta_2} \dots N_{n-1})^{\beta_{n-1}} \twoheadrightarrow (\lambda y.P)^\gamma$
- $N_n \twoheadrightarrow N'_n$

- $((\lambda y.P)^{\gamma} N'_n)^{\beta} \twoheadrightarrow \beta_n \overline{\gamma} \cdot P[\underline{\gamma} \cdot N'_n] \twoheadrightarrow (\lambda x.N)^{\alpha}$

So, by induction hypothesis,

$$h(\gamma) < h(\overline{\gamma}) \leq h(\ell(\beta_n \overline{\gamma} \cdot P[\underline{\gamma} \cdot N'_n])) \leq h(\alpha).$$

□

Lemma 5.3.23 *Any standard reduction* $M[N/x] \stackrel{st}{\twoheadrightarrow} (\lambda y.P)^{\alpha}$ *can be decomposed in one of the following two ways:*

(i) $M \twoheadrightarrow (\lambda y.M')^{\alpha}$ *and* $M'[N/x] \twoheadrightarrow P$.

(ii) $M \twoheadrightarrow M' = (\ldots((x\ M_1)^{\beta_1}\ M_2)^{\beta_2} \ldots M_n)^{\beta_n}$ *and* $M'[N/x] \twoheadrightarrow (\lambda y.P)^{\alpha}$.

Proof By induction on the length l of $M[N/x] \stackrel{st}{\twoheadrightarrow} (\lambda y.P)^{\alpha}$.

(i) $l = 0$. Then $M[N/x] = (\lambda y.P)^{\alpha}$. This means that either $M = (\lambda y.M')^{\alpha}$, and $M'[N/x] = P$, or $M = x^{\beta}$ and $x^{\beta}[N/x] = (\lambda y.P)^{\alpha}$.

(ii) $l > 0$. We shall distinguish several subcases, according to the structure of the term M.

(a) If $M = (\lambda y.M')^{\beta}$, then $M[N/x] = (\lambda y.M'[N/x])^{\beta}$. So, $\alpha = \beta$ and $M'[N/x] \twoheadrightarrow P$.

(b) If $M = (\ldots((y\ M_1)^{\beta_1}\ M_2)^{\beta_2} \ldots M_n)^{\beta_n}$, then $x = y$, since otherwise $M[N/x]$ could not reduce to a lambda abstraction. So, $M = M'$ and $M'[N/x] \twoheadrightarrow (\lambda y.P)^{\alpha}$.

(c) If $M = (\ldots((((\lambda y.A)^{\gamma}\ B)^{\beta}\ M_1)^{\beta_1}\ M_2)^{\beta_2} \ldots M_n)^{\beta_n}$, then the redex $((\lambda y.A)^{\gamma}\ B)^{\beta}$ must be contracted along the standard reduction $M[N/x] \stackrel{st}{\twoheadrightarrow} (\lambda y.P)^{\alpha}$, otherwise we could not get an abstraction as the result of this reduction. Moreover, since $((\lambda y.A)^{\gamma}\ B)^{\beta}$ is the leftmost redex in M, it is the first redex contracted in $M[N/x] \stackrel{st}{\twoheadrightarrow} (\lambda y.P)^{\alpha}$. Hence, let $Q = (\ldots(((\beta\overline{\gamma} \cdot A[\underline{\gamma} \cdot B/y])\ M_1)^{\beta_1}\ M_2)^{\beta_2} \ldots M_n)^{\beta_n}$. We have $Q[N/x] \stackrel{st}{\twoheadrightarrow} (\lambda y.P)^{\alpha}$. But the length of this derivation is $l - 1$, so we can apply the induction hypothesis. Since $M \to Q$, we conclude.

□

Lemma 5.3.24 *If* $\mathcal{P}$ *has an upper bound, and the terms* M, N *are strongly normalizing, then also* $M[N/x]$ *is strongly normalizing.*

Proof (We shall abbreviate strongly normalizable in s.n.) Let $\mathfrak{m}$ be the upper bound for the predicate $\mathcal{P}$; $\mathsf{depth}(M)$ be the length of the longest normalizing derivation of M; $\|M\|$ be the structural size of M (defined in the obvious way). The proof is by induction on the triple $\ll \mathfrak{m} - \mathfrak{h}(\ell(N)), \mathsf{depth}(M), \|M\| \gg$.
(*base case*) By hypothesis, $M = y^{\alpha}$. Then, we have two possibilities: (*i*) $y \neq x$, in which case $M[N/x] = y^{\alpha}$ is trivially s.n.; (*ii*) $x = y$, in which case, $M[N/x] = \alpha \cdot N$ is s.n., for N is s.n.
(*inductive case*) Let us classify subcases according to the size of M:

(i) $M = y^{\alpha}$. Similar to the base case. In fact, proving it we did never use the hypothesis $\mathfrak{m} - \mathfrak{h}(\ell(N)) = 0$.

(ii) $M = (\lambda y.M_1)^{\alpha}$. Then $M[N/x] = (\lambda y.M_1[N/x])^{\alpha}$. Obviously, M_1 is s.n., $\mathsf{depth}(M_1) \leq \mathsf{depth}(M)$ and $\|M_1\| < \|M\|$. So, by induction hypothesis, $M_1[N/x]$ is s.n. Thus, $M[N/x]$ is s.n.

(iii) $M = (M_1\ M_2)^{\alpha}$. Then $M[N/x] = (M_1[N/x]\ M_2[N/x])^{\alpha}$. Since $\mathsf{depth}(M_1) \leq \mathsf{depth}(M)$ and $\|M_1\| < \|M\|$, $M_1[N/x]$ is s.n. by induction hypothesis. Similarly for $M_2[N/x]$. Now, two subcases are possible:

(a) $M_1[N/x]$ never reduces to a lambda abstraction. In this case the reduction of $M[N/x]$ is the composition of two independent reductions: a reduction of $M_1[N/x]$ and a reduction of $M_2[N/x]$. So, $M[N/x]$ is s.n. since $M_1[N/x]$ and $M_2[N/x]$ are.

(b) $M_1[N/x] \twoheadrightarrow (\lambda y.P)^{\beta}$. The case in which $\mathcal{P}(\beta)$ is false is similar to the previous one. Then, let us assume that $\mathcal{P}(\beta)$ is true. In this case, $M[N/x] = (M_1[N/x]\ M_2[N/x])^{\alpha} \twoheadrightarrow ((\lambda y.P)^{\beta}\ M_2[N/x])^{\alpha}$; furthermore, $\alpha\overline{\beta} \cdot P[\underline{\beta} \cdot M_2[N/x]/y]$ is s.n. Since $M_1[N/x]$ is s.n., we can apply the standardization property of Proposition 5.3.16, getting a standard derivation $M_1[N/x] \overset{\mathsf{st}}{\twoheadrightarrow} (\lambda y.P)^{\beta}$. Then, by Lemma 5.3.23, we have the following cases:

1. $M_1 \twoheadrightarrow (\lambda y.M_3)^{\beta}$ and $M_3[N/x] \twoheadrightarrow P$. Since the term $M = (M_1\ M_2)^{\alpha}$ is s.n., also $M' = \alpha\overline{\beta} \cdot M_3[\underline{\beta} \cdot M_2/y]$ is s.n. Moreover, as $M'[N/x] = \alpha\overline{\beta} \cdot M_3[N/x][\underline{\beta} \cdot M_2[N/x]/y]$, by Lemma 5.3.4 and Lemma 5.3.5, we have $M'[N/x] \twoheadrightarrow \alpha\overline{\beta} \cdot P[\underline{\beta} \cdot M_2[N/x]/y]$. Since $M \twoheadrightarrow M'$, $\mathfrak{h}(\ell(M)) \leq \mathfrak{h}(\ell(M'))$, and $\mathsf{depth}(M') < \mathsf{depth}(M')$, we can apply induction hypothesis. That is, $M'[N/x]$ is s.n., as well as $\alpha\overline{\beta} \cdot P[\underline{\beta} \cdot M_2[N/x]/y]$.

2. $M_1 \twoheadrightarrow M_1' = (\ldots((x^\gamma A_1)^{\alpha_1} A_2)^{\alpha_2} \ldots A_n)^{\alpha_n}$ for some M_1' such that $M_1'[N/x] \twoheadrightarrow (\lambda y.P)^\beta$. Since $M_1[N/x]$ is s.n., also P is s.n. Moreover:

   $$M_1'[N/x] = (\ldots((\gamma{\cdot}N)\ A_1[N/x])^{\alpha_1} \ldots A_n[N/x])^{\alpha_n}.$$

   By Lemma 5.3.21, $h(\ell(\gamma \cdot N)) \leq h(\beta)$. So, we have:

   $$h(\ell(N)) \leq h(\ell(\gamma{\cdot}N)) \leq h(\beta) < h(\underline{\beta}) \leq h(\ell(\underline{\beta}{\cdot}M_2[N/x]))$$

   Then, by induction hypothesis, $\alpha\overline{\beta}{\cdot}P[\underline{\beta}{\cdot}M_2[N/x]/y]$ is s.n.

□

Proposition 5.3.25 *If $\mathcal{P}$ has an upper bound, then every term M is strongly normalizable.*

Proof By structural induction on M.

(i) $M = x^\alpha$. Trivial.

(ii) $M = (\lambda y.M_1)^\alpha$. M_1 is strongly normalizing by induction hypothesis, and so is M.

(iii) $M = (M_1\ M_2)^\alpha$. By induction hypothesis, M_1 and M_2 are strongly normalizing. If M_1 does never reduce to a lambda abstraction the result is obvious. Otherwise, suppose $M_1 \twoheadrightarrow (\lambda x.M_3)^\beta$; we must prove that $\alpha\overline{\beta} \cdot M_3[\underline{\beta} \cdot M_2/x]$ is strongly normalizing. Since M_3 and M_2 are strongly normalizing, so are $\alpha\overline{\beta} \cdot M_3$ and $\underline{\beta} \cdot M_2$. Thus, by Lemma 5.3.24 we conclude.

□

Theorem 5.3.26 (Church-Rosser) *If $\sigma : M \twoheadrightarrow N$ and $\rho : M \twoheadrightarrow P$, then there exists a term Q such that $N \twoheadrightarrow Q$ and $P \twoheadrightarrow Q$.*

Proof Let $\sigma : M = M_0 \xrightarrow{R_1} M_1 \xrightarrow{R_2} \ldots \xrightarrow{R_n} M_n = N$ and $\rho : M = M_0 \xrightarrow{S_1} P_1 \xrightarrow{S_2} \ldots \xrightarrow{S_m} P_m = P$. Let $\mathcal{P}_\sigma$ and $\mathcal{P}_\rho$ be the predicates defined in the following way:

(i) $\mathcal{P}_\sigma(\alpha)$ if and only if $\exists i, 1 \leq i \leq n, \alpha = \mathsf{degree}(R_i)$;

(ii) $\mathcal{P}_\rho(\alpha)$ if and only if $\exists j, 1 \leq j \leq m, \alpha = \mathsf{degree}(S_j)$.

Let $\mathcal{P} = \mathcal{P}_\sigma \bigcup \mathcal{P}_\rho$. Obviously, $\mathcal{P}$ has an upper bound, and since the two reductions σ and ρ are legal for $\mathcal{P}$, M is strongly normalizable (with respect to P). Then, by Proposition 5.3.10, we conclude. □

Theorem 5.3.27 (standardization) *If* $M \twoheadrightarrow N$, *then* $M \overset{st}{\twoheadrightarrow} N$.

Proof Similar to the previous one, using Proposition 5.3.16 in the place of Proposition 5.3.10. □

Exercise 5.3.28 (Difficult) Prove that if $\sigma : M \twoheadrightarrow N$ and $\rho : M \twoheadrightarrow P$, then there exists a term Q such that $\rho/\sigma : N \twoheadrightarrow Q$ and $\sigma/\rho : P \twoheadrightarrow Q$.

Exercise 5.3.29 (Difficult) Prove that if $\rho : M \twoheadrightarrow N$, then there exists a standard reduction $\rho_s : N \overset{st}{\twoheadrightarrow} Q$ such that $\rho \equiv \rho_s$.

Remark 5.3.30 In the prove of Church-Rosser property (Theorem 5.3.26) we have not explicitly seen as closing reduction are done. Anyhow, the previous exercise shows that, similarly to what done proving local confluence, the diamond is closed by the image of the reduction on the opposite side. Furthermore, let us note that this property is in some sense encoded into the proof of Theorem 5.3.26: the predicate $\mathcal{P}$ that we choose is exactly the one that enables families that were present in the initial reductions ρ and σ.

5.3.2 Labeled and unlabeled λ-calculus

We shall now prove one of the most interesting results of the labeled λ-calculus. As we already noted studying families, in the unlabeled case there are pairs of reductions starting and ending with the same pair of terms that cannot be considered equivalent (remind the example $(I(Ix))$). We already pointed out with an example that this is not the case in the labeled calculus. In this section we will prove that this is a general property of labeled λ-calculus. In fact, we will see that given two terms M and N such that $M \twoheadrightarrow N$, then there is a unique standard reduction $M \overset{st}{\twoheadrightarrow} N$.

Definition 5.3.31 Given a labeled term M, we call $\tau^{-1}(M)$ the corresponding unlabeled term obtained by erasing all labels. Formally:

(i) $\tau^{-1}(x^\alpha) = x$
(ii) $\tau^{-1}((\lambda x.M)^\alpha) = \lambda x.\tau^{-1}(M)$
(iii) $\tau^{-1}((M\ N)^\alpha) = (\tau^{-1}(M)\ \tau^{-1}(N))$

We shall now introduce a new measure for labels that will become useful in proving the next proposition.

Definition 5.3.32 The *size* $\|\alpha\|$ of a label α is the sum of the total number of letters, overlinings and underlinings in α. Formally:

(i) $\|a\| = 1$ if a is an atomic label;
(ii) $\|\alpha\beta\| = \|\alpha\| + \|\beta\|$;
(iii) $\|\overline{\alpha}\| = |\underline{\alpha}| = \|\alpha\| + 1$.

Lemma 5.3.33 *If* $M \twoheadrightarrow M'$, *then* $\|\ell(M)\| \leq \|\ell(M')\|$.

Proof Immediate. □

Proposition 5.3.34 *For every pair of labeled terms* U, V *such that* $U \twoheadrightarrow V$, *there exists exactly one standard reduction* $U \overset{st}{\twoheadrightarrow} V$.

Proof If $U \twoheadrightarrow V$, by the standardization theorem there exists at least one standard reduction $\sigma : U \twoheadrightarrow V$. We must prove that this is unique. The proof is by induction on the length l of σ and the structure of U.
(*base case*) $U = V = x^{\alpha}$. Then the only standard reduction is the empty one.
(*inductive case*)

(i) $U = x^{\alpha}$. This case has been already considered.
(ii) $U = (\lambda x.U_1)^{\alpha}$. Then $V = (\lambda x.V_1)^{\alpha}$ and $U_1 \overset{st}{\twoheadrightarrow} V_1$ with a standard reduction σ' of length l. By induction hypothesis σ' is unique, and so is σ.
(iii) $U = (U_1\ U_2)^{\alpha}$. σ must be either of the kind

$$U = (U_1\ U_2)^{\alpha} \overset{st}{\twoheadrightarrow} (V_1\ U_2) \overset{st}{\twoheadrightarrow} (V_1\ V_2)^{\alpha} = V$$

or

$$U = (U_1\ U_2)^{\alpha} \overset{norm}{\twoheadrightarrow} ((\lambda x.U_3)^{\beta}\ U_2) \to \alpha\overline{\beta} \cdot U_3[\underline{\beta} \cdot U_2/x] \overset{st}{\twoheadrightarrow} V$$

where $(\lambda x.U_3)^{\beta} = \mathsf{Abs}(U_1)$.
We prove first that, in the labeled system, *all* standard reductions between U and V are of a same kind. Indeed, suppose we might have two standard reductions ρ and τ of different kinds. If ρ would be of the first kind above, we would also have $\ell(V) = \alpha = \ell(U)$. On the other side, if τ would be of the second kind, by Lemma 5.3.33, we would have

$$\|\ell(U)\| = \|\alpha\| < \|\ell(\alpha\overline{\beta} \cdot U_3[\underline{\beta} \cdot U_2/x])\| \leq \|\ell(V)\|$$

and thus $\ell(U) \neq \ell(V)$.

Coming back to the proof of the proposition, we can thus distinguish two subcases, according to the kind of the standard reduction σ.

(a) if σ is of the first kind, the result follows by induction, since the standard reductions $U_1 \overset{st}{\twoheadrightarrow} V_1$ and $U_2 \overset{st}{\twoheadrightarrow} V_2$ are unique.

(b) If σ is of the second kind, every other standard reduction must be of the same kind. Since the initial part of the derivation is normal, this must be common to every such reduction. Since, by induction on the length of the derivation, there exists a unique standard reduction from $\alpha\overline{\beta} \cdot U_3[\underline{\beta} \cdot U_2/x]$ to V, σ is unique too.

□

Note that the previous proposition does not hold in the (unlabeled) λ-calculus (take for instance $U = (I(I\,x))$ and $V = (I\,x)$). We also invite the reader to not confuse the previous property with the one that Theorem 5.3.27 induces on the unlabeled λ-calculus. In fact, in the unlabeled calculus we can still say that given a reduction $\rho : M \twoheadrightarrow N$, there is a unique standard reduction $\rho_s : M \overset{st}{\twoheadrightarrow} N$ equivalent to it (see also Exercise 5.3.29). Nevertheless, two non-equivalent reductions connecting the same pair of terms lead to two distinct standard reduction. This is no more true in the labeled calculus where two reductions are equivalent if and only if connect the same pair of terms (see next theorem). In a sense, the labeled λ-calculus *does not make syntactical mistakes* identifying terms coming from "different" derivations. This intuition is formalized by the following theorem.

Theorem 5.3.35 *Let U be a labeled λ-term with $M = \tau^{-1}(U)$. Let $\sigma : M \twoheadrightarrow N$ and $\rho : M \twoheadrightarrow P$. Consider the labeled reductions $\sigma_1 : U \twoheadrightarrow V$ and $\rho_1 : U \twoheadrightarrow W$ respectively isomorphic to σ and ρ. Then:*

(i) $\sigma \equiv \rho$ *if and only if* $V = W$

(ii) $\sigma \sqsubseteq \rho$ *if and only if* $V \twoheadrightarrow W$.

Proof

(i) Let σ_1' and ρ_1' be the standard reductions corresponding to σ_1 and ρ_1, respectively. Then, $\sigma \equiv \tau$ if and only if $\sigma_1 \equiv \tau_1$ if and only if $\sigma_1' = \tau_1'$. But for the previous proposition, $\sigma_1' = \tau_1'$ if and only if $V = W$.

(ii) By definition, $\sigma \sqsubseteq \rho$ if and only if there exists τ such that $\sigma\tau \equiv \rho$. Let $\tau_1 : V \twoheadrightarrow T$ be the labeled reduction isomorphic to τ. By the previous item, $\sigma\tau \equiv \rho$ if and only if $T = W$. So, $\sigma \sqsubseteq \tau$ if and only if $V \twoheadrightarrow W$.

□

5.3.3 Labeling and families

Proposition 5.3.36 *If* $M \twoheadrightarrow N$ *in the labeled lambda calculus, and* S *is a redex in* M*, then all residuals of* S *in* N *have the same degree of* S*.*

Proof Let us start with the case $M \xrightarrow{R} N$. Let $R = (\lambda x.A)^{\alpha}\ B)^{\beta}$ and $S = (\lambda y.C)^{\gamma}\ D)^{\delta}$. The proof is by cases, according to the mutual positions of R and S in M.

(i) R and S are disjoint. Trivial.

(ii) If S contains R in C (respectively D), then the (unique) residual of S in N is of the form $(\lambda y.C')^{\gamma}\ D)^{\delta}$ (respectively $(\lambda y.C)^{\gamma}\ D')^{\delta}$, which has the same degree of S.

(iii) If R contains S in A, then S has a unique residual in N of the form $(\lambda y.C[\underline{\alpha} \cdot B/x])^{\gamma}\ D[\underline{\alpha} \cdot B/x])^{\delta'}$, where only the external label δ can be modified into δ' (this happens when $A = (\lambda y.C)^{\gamma}\ D)^{\delta}$).

(iv) If R contains S in B, then all residuals of the redex S in N are of the form $(\lambda y.C)^{\gamma}\ D)^{\delta'}$, where only the external label δ can be modified into δ' (this happens when $B = (\lambda y.C)^{\gamma}\ D)^{\delta}$).

□

Exercise 5.3.37 Reformulate the proof of the previous proposition in terms of labeled syntax trees. Note in particular that the four cases in the proof correspond in order to: (*i*) the edge of the redexes appear in disjoint subtrees; (*ii*) the edge of R is in the subtree of the application corresponding to S; (*iii*) the edge of S is in the body of the abstraction of R; (*iv*) the edge of S is in the subtree of the argument of R. In particular, note that the last one is the only case in which the edge of R is duplicated.

Proposition 5.3.38 *If* $M \xrightarrow{R} N$ *and* S *is a redex in* N *created by* R *then* $\|\mathsf{degree}(R)\| < \|\mathsf{degree}(S)\|$*.*

Proof Let $R = (\lambda x.A)^{\alpha}\ B)^{\beta}$. Three cases are possible:

Upward creation. In M there is a subterm of the form $(((\lambda x.A)^{\alpha}\ B)^{\beta}\ D)^{\delta}$ where $A = (\lambda y.C)^{\gamma}$. Then, $S = ((\beta\overline{\alpha} \cdot A[\underline{\alpha} \cdot B/x])\ D)^{\delta} = ((\lambda y.C[\underline{\alpha} \cdot B/x])^{\beta\overline{\alpha}\gamma}\ D)^{\delta}$ and $\|\alpha\| < \|\beta\overline{\alpha}\gamma\|$.

Downward creation. In A there is a subterm of the form $((x)^{\epsilon}\ D)^{\delta}$ and $B = (\lambda y.C)^{\gamma}$. Then, $S = ((x)^{\epsilon}\ D)^{\delta}[\underline{\alpha} \cdot (\lambda y.C)^{\gamma}/x] = ((\lambda y.C)^{\epsilon\underline{\alpha}\gamma}\ D)^{\delta}$, and $\|\alpha\| < \|\epsilon\underline{\alpha}\gamma\|$.

Identity. This is a combination of the two previous cases. Namely, we have a subterm in M of the form $(((\lambda x.A)^{\alpha}\ B)^{\beta}\ D)^{\delta}$ where $A = x^{\epsilon}$ and $B = (\lambda y.C)^{\gamma}$. Then, $S = ((\beta\overline{\alpha} \cdot A[\underline{\alpha} \cdot B/x])\ D)^{\delta} = ((\lambda y.C)^{\beta\overline{\alpha}\epsilon\underline{\alpha}\gamma}\ D)^{\delta}$ and $\|\alpha\| < \|\beta\overline{\alpha}\epsilon\underline{\alpha}\gamma\|$.

□

Definition 5.3.39 (INIT) We shall say that the predicate $\mathsf{INIT}(M)$ is verified by a labeled term M if and only if the labels of all subterms of M are atomic and pairwise distinct.

Proposition 5.3.40 *Let M be a term such that $\mathsf{INIT}(M)$ holds. For any reduction $M \twoheadrightarrow N$, a redex S in N is a residual of a redex R in M if and only if they have the same degree.*

Proof The only if direction is Proposition 5.3.36. For the if direction, we proceed by induction on the length l of the reduction $M \twoheadrightarrow N$. If $l = 0$, the result follows by the conditions imposed by $\mathsf{INIT}(M)$. If $l > 0$, let $M \twoheadrightarrow N' \xrightarrow{P} N$. By hypothesis, there exists a redex R in M with the same degree of S. Since, by $\mathsf{INIT}(M)$, all redexes in M have atomic degree, the degree of S must be atomic too. This means that S cannot be created by the firing of P, since otherwise $\|\mathsf{degree}(S) > 1\|$. So, S must be the residual of a redex S' in M' with the same degree of R. By induction hypothesis S' is a residual of R, and so is S. □

Theorem 5.3.41 *Let ρR and σS be two redexes with histories $\rho : M \twoheadrightarrow N$ and $\sigma : M \twoheadrightarrow P$. Let us take the corresponding isomorphic reductions $\rho_1 : U \twoheadrightarrow V$ and $\sigma_1 : U \twoheadrightarrow W$ in the labeled system (the initial labeling of U can be arbitrary). If $\rho R \simeq \sigma S$, then R and S have the same degree (in V and W, respectively).*

Proof This is an easy corollary of Theorem 5.3.35 and Proposition 5.3.36. □

Let us remark again that the previous theorem holds for any labeling of the initial term M. According to the intended interpretation of labeling as the name associated to all the edges with a same origin, the previous theorem states that this interpretation is sound. In fact, all redexes in the same family with respect to a given initial term are marked by the same label. Furthermore, the independence from initial labeling state that to equate objects in the initial term does not create troubles. For instance, the term might be the result of a previous computation and according to this labels might denote sharability of some objects. More technically, this compositionality od the property is way it can be proved inductively on the length of the derivation.

In Chapter 6 we shall prove the converse of Theorem 5.3.41, but under the (essential) assumption that $\mathsf{INIT}(U)$ holds. In fact, according to the idea that all redexes in the initial term are in different families and then unsharable, all the edges of the initial term must be marked with different (atomic) labels—it other words, the initial term is not the result of some previous computation or equivalently, we do not its history. Under this assumption, we will be able to prove that two redexes are in the same family only if they have the same degree. Let us note that Proposition 5.3.40 is not enough to get this equivalence between the equivalence families and degree. In fact, Proposition 5.3.40 proves the result in the case of atomic labels, but does not allow to immediately extend it to the case of composite degrees. Let us compare this with the situation in extraction relation. Also in that case it is immediate to prove that when a redex R has an ancestor S in the initial term, then S is the canonical representative of R plus its history (see Lemma 5.2.8). The difficult part in proving uniqueness of a canonical representative is when the redex is created along the reduction (see the rest of section 5.2). In the labeled case, the difficult part will be to show the uniqueness of this canonical representative in terms of labels, i.e., that two canonical representatives for extraction cannot have the same degree.

5.4 Reduction by families

In this section we aim at finding the syntactic counterpart of an evaluator never duplicating redexes, according to notion of duplication induced by family relation (i.e., the zig-zag relation defined in Section 5.1). To this aim we introduce a strategy of derivation by families, a parallel reduction in which at each step several redexes in the same family can be reduced in parallel. The idea is that reducing at once all the redexes

in a given families, in the future, we cannot evaluate any redex in this family. In fact, once a family has been completely erased from a term, it is impossible it can reappear along the reduction (let us note that this follows from the interpolation property of Lemma 5.1.12).

In order to formalize these derivations, we foremost generalize the finite development theorem.

Definition 5.4.1 Let $[\rho R] = \{\sigma S \mid \sigma S \simeq \rho R\}$ be the *family class* of ρR. Let $\rho = \mathcal{F}_1 \cdots \mathcal{F}_n$. Then,

$$\mathsf{FAM}(\rho) = \{[\mathcal{F}_1 \cdots \mathcal{F}_i R] \mid R \in \mathcal{F}_{i+1}, i = 0, 1, \ldots, n-1\}$$

is the set of family classes contained in ρ.

Definition 5.4.2 (development of family classes) Let $\mathcal{X}$ be a set of family classes. A derivation ρ is *relative* to $\mathcal{X}$ if $\mathsf{FAM}(\rho) \subseteq X$. A derivation ρ is a *development* of $\mathcal{X}$ if there is no redex R such that $[\rho R] \in \mathcal{X}$.

Theorem 5.4.3 (generalized finite developments) *Let $\mathcal{X}$ be a (finite) set of family classes. Then:*

(i) *There is no infinite derivation relative to $\mathcal{X}$.*

(ii) *If ρ and σ are two developments of $\mathcal{X}$ then, $\rho \equiv \sigma$.*

Proof

(i) Let $\mathcal{X}$ be a finite set of family classes with respect to the initial term M. Let us assume that $\mathsf{INIT}(M)$ is true. Let $\quad = \{\mathsf{degree}(\rho R) \mid [\rho R] \in \mathcal{X}\}$ and $\mathcal{P} = \{\alpha \mid h(\alpha) \leq \max\{h(\beta) \mid \beta \in \quad\}\}$. By Theorem 5.3.41, any family class has a unique degree. Hence, $\quad$ is finite and $\mathcal{P}$ has an upper bound. By Proposition 5.3.25, any labeled derivation legal for $\mathcal{P}$ is finite. Since any derivation relative to $\mathcal{X}$ is legal for $\mathcal{P}$, we conclude.

(ii) Let us observe that, if ρ and σ are relative to $\mathcal{X}$, then $\rho \sqcup \sigma$ is also relative to $\mathcal{X}$. Now, by definition, if ρ and σ are two developments of $\mathcal{X}$, then $\rho \equiv \rho \sqcup \sigma$ and $\sigma \equiv \rho \sqcup \sigma$. Thus, $\rho \equiv \sigma$ by transitivity.

□

Let us now prove our claim that, after the development of a set of families $\mathcal{X}$, no redex in a family contained in $\mathcal{X}$ can be created anymore along the reduction. The main lemma is preceded by a more technical property useful for its proof.

Lemma 5.4.4 *Let $\rho\mathsf{R}$ be the canonical derivation of $\sigma\mathsf{S}$. Then $\rho \sqsubseteq \sigma$ if and only if $\rho\mathsf{R} \leq \sigma\mathsf{S}$.*

Proof The if direction follows by definition of copy relation. Let us focus on the only-if direction. Let τ be the standard derivation of σ. Then $\tau\mathsf{S} \rhd \rho\mathsf{R}$ and $\rho \sqsubseteq \tau$, since $\tau \equiv \sigma$. Furthermore, it suffices to prove $\rho\mathsf{R} \leq \tau\mathsf{S}$. We proceed by induction on $|\tau|$.
If $\tau = \emptyset$, then $\mathsf{S} \rhd \rho\mathsf{R}$ implies $\rho = \emptyset$ and $\mathsf{R} = \mathsf{S}$. Thus, $\rho\mathsf{R} \leq \tau\mathsf{S}$.
Let $\tau = \mathsf{T}\tau'$. Since both ρ and τ are standard, there are two cases:

(i) $\rho = \mathsf{T}\rho'$ and $\rho'\mathsf{R}$ is the canonical derivation of $\tau'\mathsf{S}$. Then $\rho \sqsubseteq \tau$ implies $\rho' \sqsubseteq \tau'$ by left cancellation. Therefore, $\rho\mathsf{R} \leq \tau\mathsf{S}$ because, by induction, $\rho'\mathsf{R} \leq \tau'\mathsf{S}$.

(ii) $\mathsf{T}\rho'\mathsf{R}' \rhd \rho\mathsf{R}$, where $\rho'\mathsf{R}'$ is the canonical derivation of $\tau'\mathsf{S}$. Then $\rho' \sqsubseteq \rho/\mathsf{T}$, by definition of $\rhd$. This fact and $\rho \sqsubseteq \tau$ imply $\rho/\mathsf{T} \sqsubseteq \tau/\mathsf{T} = \tau'$. Therefore, $\rho' \sqsubseteq \rho/\mathsf{T} \sqsubseteq \tau'$ and, by induction, $\rho'\mathsf{R}' \leq \tau'\mathsf{S}$. By the interpolation property (Lemma 5.1.12), there is S' such that $\rho'\mathsf{R}' \leq (\rho/\mathsf{T})\mathsf{S}' \leq \tau'\mathsf{S}$. Then, $\mathsf{T}(\rho/\mathsf{T})\mathsf{S}' \leq (\mathsf{T}\tau')\mathsf{S} = \tau\mathsf{S}$. Moreover, $\rho(\mathsf{T}/\rho)\mathsf{S}' \leq \tau\mathsf{S}$, for $\mathsf{T}(\rho/\mathsf{T}) \equiv \rho(\mathsf{T}/\rho)$. Finally, $\mathsf{S} \in \mathsf{R}/(\mathsf{T}/\rho)$, since $\rho\mathsf{R}$ is canonical and $\rho(\mathsf{T}/\rho)\mathsf{S}' \simeq \rho\mathsf{R}$. Thus, $\rho\mathsf{R} \leq \tau\mathsf{S}$.

□

Lemma 5.4.5 *Let ρ be a development of $\mathcal{X}$.*

(i) *Let $\sigma\mathsf{S}$ be the canonical derivation of $\rho\mathsf{R}$. Then $\sigma\mathsf{S} \leq \rho\mathsf{R}$.*

(ii) *For every $\sigma\mathsf{S}$ such that $\rho \sqsubseteq \sigma$, $[\sigma\mathsf{S}] \notin \mathsf{FAM}(\rho)$.*

Proof

(i) By hypothesis, σ is relative to $\mathcal{X}$. Then $\sigma \sqsubseteq \rho$, since σ can be always extended to a development $\sigma\tau$ of $\mathcal{X}$ and $\sigma\tau \equiv \rho$, by Theorem 5.4.3. Thus $\sigma\mathsf{S} \leq \rho\mathsf{R}$, by Lemma 5.4.4.

(ii) By contradiction. Let $\sigma\mathsf{S} \in \mathsf{FAM}(\rho)$, $\rho = \rho_1 \mathcal{F} \rho_2$ and $\mathsf{R} \in \mathcal{F}$ such that $\rho_1\mathsf{R} \simeq \sigma\mathsf{S}$. Let $\rho'\mathsf{R}'$ be the canonical derivation of $\rho_1\mathsf{R}$ and $\sigma\mathsf{S}$. By definition, ρ' is relative to $\mathsf{FAM}(\rho)$. Therefore, $\rho' \sqsubseteq \rho$ and, by transitivity, $\rho' \sqsubseteq \sigma$. Therefore, $\rho'\mathsf{R}' \leq \sigma\mathsf{S}$, by Lemma 5.4.4. By the interpolation property (Lemma 5.1.12), there exists T such that $\rho'\mathsf{R}' \leq \rho\mathsf{T} \leq \sigma\mathsf{S}$. But this contradicts this contradicts the hypothesis that ρ is a development of $\mathcal{X}$, since under this hypothesis ρ should also be a development of $\mathsf{FAM}(\rho)$.

□

5.4.1 Complete derivations

We can finally define the reductions by families we are interested in.

Definition 5.4.6 (complete derivation) A derivation $\mathcal{F}_1 \cdots \mathcal{F}_n$ is *complete* if and only if $\mathcal{F}_i \neq \emptyset$ and $\mathcal{F}_i$ is a maximal set of redexes such that

$$\forall R, S \in \mathcal{F}_i.\ \mathcal{F}_1 \cdots \mathcal{F}_{i-1} R \simeq \mathcal{F}_1 \cdots \mathcal{F}_{i-1} S$$

for $i = 1, 2, \ldots, n$.

The following lemma shows that complete derivations are particular developments.

Lemma 5.4.7 *Every complete derivation ρ is a development of* $\mathsf{FAM}(\rho)$.

Proof By induction on $|\rho|$. The base case is obvious. Let $\rho = \sigma\mathcal{F}$, where $\mathcal{F}$ is a maximal set of redexes in the same family. Then, by inductive hypothesis, σ is a development of $\mathsf{FAM}(\sigma)$. Therefore, by Lemma 5.4.5(2), there is no $\sigma' S$, $\sigma \sqsubseteq \sigma'$, such that $[\sigma' S] \in \mathsf{FAM}(\sigma)$. By contradiction, let us assume that there exists R such that $\rho R \simeq \sigma S$. Namely, let us assume that ρ is not a development of $\mathsf{FAM}(\rho)$. Let $\tau S'$ be the canonical derivation of σS. Then $\tau \sqsubseteq \sigma \sqsubseteq \rho = \sigma\mathcal{F}$. Moreover, by Lemma 5.4.5(1), $\tau S' \leq \rho R$ and, by the interpolation lemma (Lemma 5.1.12), there exists T such that $\sigma T \leq \rho R$. This means that $\sigma T \simeq \sigma S$ with $R \in T/\mathcal{F}$, invalidating the hypothesis that $\mathcal{F}$ is a maximal set of redexes in the same family. □

Complete derivations contract *maximal* set of copies of a single redex. When complete derivations are considered, this means that deciding if two redexes are in the same family, may be safely reduced to check the copy-relation.

Lemma 5.4.8 *A derivation $\rho = \mathcal{F}_1 \cdots \mathcal{F}_n$ is complete if and only if, for $i = 1, \ldots, n$, $\mathcal{F}_i$ is a* maximal set of copies. *Namely, for any i, there exist $\sigma_i S_i$ and τ_i such that $\sigma_i \tau_i \equiv \rho_i$ and $\mathcal{F}_i = S/\tau_i$.*

Proof

(*only-if direction*) Let ρ be a complete reduction and R be a redex with history ρ. Let $\mathcal{F}$ be the set of redexes T such that $\rho R \simeq \rho T$. Let $\mathcal{F}'$ be a set of redexes containing R and such that there exist σS and τ with $\sigma\tau \equiv \rho$ and $\mathcal{F}' = S/\tau$. We shall prove that $\mathcal{F} = \mathcal{F}'$.

Surely, $\mathcal{F}' \subseteq \mathcal{F}$, by definition of $\simeq$. By the completeness of ρ and Lemma 5.4.7, ρ is a development of $\mathsf{FAM}(\rho)$. Let $\rho'R'$ be the canonical derivation of ρR. Then $\rho'R' \leq \rho R$, by Lemma 5.4.5(1). Therefore, for every $S \in \mathcal{F}$, $\rho'R' \leq \rho S$; which means $\mathcal{F} \subseteq \mathcal{F}'$, for $\mathcal{F}'$ is maximal.

(*if direction*) By contradiction. Let us assume that ρ is complete and that there exists i such that $\mathcal{F}_i$ is not a maximal set of copies. Let $R \in \mathcal{F}_i$. Therefore, there is $S \notin \mathcal{F}_i$ such that, for some τT, $\tau T \leq \mathcal{F}_1 \cdots \mathcal{F}_{i-1} R$ and $\tau T \leq \mathcal{F}_1 \cdots \mathcal{F}_{i-1} S$. Then, $\mathcal{F}_1 \cdots \mathcal{F}_{i-1} R \simeq \mathcal{F}_1 \cdots \mathcal{F}_{i-1} S$, which invalidates the hypothesis that $\mathcal{F}_i$ is a maximal set of redexes in the same family.

□

Exercise 5.4.9 Prove that for any reduction $\rho : M \twoheadrightarrow N$ there exists a complete reduction ρ_c equivalent to ρ (i.e., $\rho_c \equiv \rho$).

Proposition 5.4.10 *Let ρ be a complete derivation. Then $|\rho| = \sharp(\mathsf{FAM}(\rho))$, where $\sharp(\mathsf{FAM}(\rho))$ is the cardinality of $\mathsf{FAM}(\rho)$.*

Proof Easy consequence of Lemma 5.4.7 and of the requirement that the steps of complete derivations are non-empty. □

Reasoning in graphs, it will be useful to have names for each link participating in a β-reduction.

Definition 5.4.11 In β-reduction, the links that are consumed by and are created reducing a redex will be referred to as indicated in Figure 5.8.

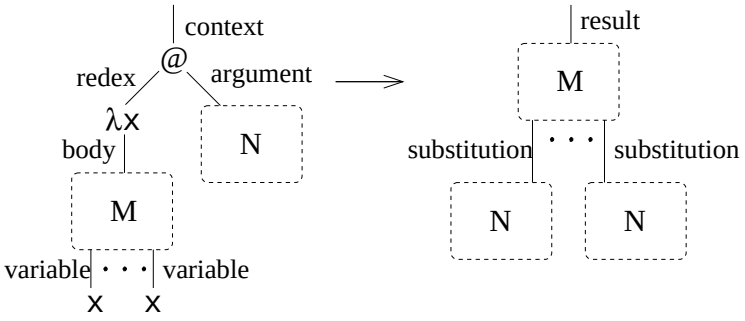


Fig. 5.8. Links involved in β-reduction.

Lemma 5.4.12 *Let $\mathcal{F}_1 \cdots \mathcal{F}_n$ be a complete derivation. Then:*

(i) *Each new link created by a (parallel) reduction is marked with a label that did not previously appear in the expression.*

(ii) *The labels on result links created by $\mathcal{F}_i$ are different from the labels on substitution links. The labels on two result links created by $\mathcal{F}_i$ are identical if and only if the labels on the antecedent context, redex and body links are respectively equal. The labels on two substitution links created by $\mathcal{F}_i$ are identical if and only if the labels on the antecedent variable, redex and argument links are respectively equal.*

Proof Easy, by definition of labeling and complete derivation. □

5.5 Completeness of Lamping's algorithm

Lévy's work concluded in 1980 leaving open the issue of designing a λ-evaluator implementing complete reductions. This issue slept ten years before Lamping awakened it presenting its evaluator in 1989.

Having the formalization of Lévy optimality at hand we can now prove that the algorithm presented in Chapter 3—a simplified version of Lamping's one—fits Lévy's completeness requirement. Most of the definitions and results in this section are due to Lamping [Lam89].

In order to prove that Lamping's algorithm is complete, it suffices to show that all the redexes in a maximal set have a unique representation in the sharing graphs. Let us be more precise on this point.

Let G be a sharing graph obtained along the reduction of $[M]$. Let T be the λ-term that matches with G (see Definition 3.5.13). Correctness implies that $\rho : M \twoheadrightarrow T$ (see Theorem 3.5.15). Furthermore, let ρ_c be a complete reduction equivalent to ρ (see Exercise 5.4.9), we have to prove that in G all the redexes in a same family have a unique representation, in this way contracting such a shared redex we would contract a maximal set of redexes. In terms of labels, to prove completeness means to show that any β-redex edge in G represents *only and all* β-redexes of T with the same label.

The proof will be pursued exploiting the latter correspondence of labels. Nevertheless, it is not trivial, for the correspondence between edges and connections in the sharing graph is not a function from labels to identity connections (i.e., sequences of edges crossing fans or brackets only). Namely, uniqueness of representation might not hold for edges with the same label that are not β-redexes.

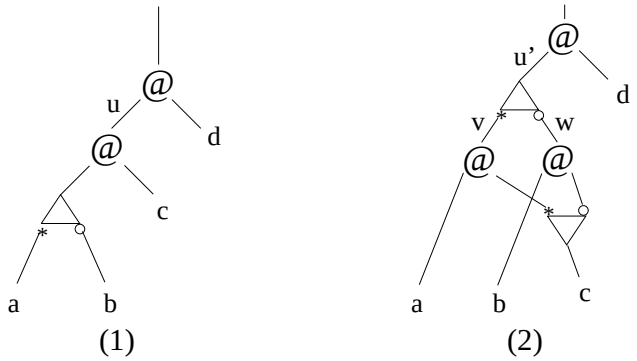


Fig. 5.9. Duplication of edges

For instance, let us consider the example in Figure 5.9. The edge marked u in the sharing graph in Figure 5.9(1) represents a set of edges in the syntax tree associated to the graph, all with the same label. However, as soon as the fan-@ interaction is contracted (see Figure 5.9(2)), the edge u is split into the two (distinct) paths $u' \cdot v$ and $u' \cdot w$. The representations of these paths in the syntax tree are the same of u, since a fan-@ interaction does not change the syntax tree matching the sharing graph. Therefore, $u' \cdot v$ and $u' \cdot w$ represent edges marked by the same label, even if they are different. Nevertheless, β-interactions involving the topmost application are not copies, so they would not get the same label. But this is definitely the case, as such reduction will eventually apply two intrinsically different functions: the one resulting from the contraction of the bottom-left application, and the other one obtained by evaluating the bottom-right application. Let us see this point in full details. Let $@^*$ and $@^\bullet$ be respectively the topmost and bottommost redexes in Figure 5.9(1). Let α be the label of the edges corresponding to u in Figure 5.9(1). According to our previous reasonings, both $u' \cdot v$ and $u' \cdot w$ corresponds to edges with label α. Furthermore, propagating the fan in Figure 5.9(2) through $@^*$, we eventually get two instances of the edge u, say u_1 and u_2, connecting pair of @-nodes that are instances of $@^*$ and $@^\bullet$ (i.e., for $i = 1,2$, the edge u_i connects a node $@_i^*$ to a node $@_i^\bullet$, where $@_i^*$ and $@_i^\bullet$ are instances of $@^*$ and $@^\bullet$, respectively). Again, we know that u_1 and u_2 represent edges with label α. Now, the edges u_1 and u_2 may become part of a redex in the sharing graph only after firing redexes involving the corresponding bottom nodes. Moreover, by induction hypothesis, we may assume that such two redexes yield

distinct labels β_1 and β_2. Summarizing, any redex involving $@_1^*$ will yield a label $\alpha\overline{\beta_1}\gamma_1$, while any redex involving $@_2^*$ will yield a label $\alpha\overline{\beta_2}\gamma_2$, that are definitely distinct.

In a sense the evaluator goes ahead duplicating edges yielding the same label that will participate to the creation of different (families of) β-redexes.

Unfortunately, the reasoning applied in the example does not work for β-rule. In this case, new redexes can be created, all involving the β-redex fired by the rule. Because of this, the induction hypothesis must be stronger than the one used above. More precisely, we need to characterize those segments in the syntax tree that have a unique representation in the corresponding sharing graph.

Definition 5.5.1 (prerequisite chain) Let $\mathtt{n}$ be a node in the syntax tree of a λ-term. The *prerequisite chain* φ of $\mathtt{n}$ is (if any) the unique sequence of edges (a path assuming that variables are back-connected to their binders) according to the following inductive definition by cases on the type of $\mathtt{n}$:

(i) $\mathtt{n}$ *is a* λ*-node*: Let ν be the context edge of $\mathtt{n}$. If ν is a β-redex, then $\varphi = \nu$, otherwise $\varphi = \nu \cdot \psi$, where ψ is the prerequisite chain of the node above $\mathtt{n}$.

(ii) $\mathtt{n}$ *is an* @*-node*: Let ν be the function edge of $\mathtt{n}$. If ν is a β-redex, then $\varphi = \nu$, otherwise $\varphi = \nu \cdot \psi$, where ψ is the prerequisite chain of the node connected to the function port of $\mathtt{n}$.

(iii) $\mathtt{n}$ *is a variable*: Then φ is the prerequisite chain of the λ-node binding $\mathtt{n}$.

For instance, let us consider the syntax tree in Figure 5.10. The vertex λx has no prerequisite chain; the prerequisite chain of the node λy is the edge above it; the prerequisite chain of the node λv is the edge above it followed by the β-redex. Finally, let us compute the prerequisite chain of the rightmost application (the chain drawn by a dashed line in Figure 5.10): the prerequisite chain of the topmost application is its function edge followed by the β-redex (the short dotted line); the prerequisite chain of the node λu is its context edge followed by the prerequisite chain of the topmost application (the curved dotted line); therefore, the prerequisite chain of the rightmost application is its function edge followed by the prerequisite chain of the variable u, which is the same of the prerequisite chain of λu.

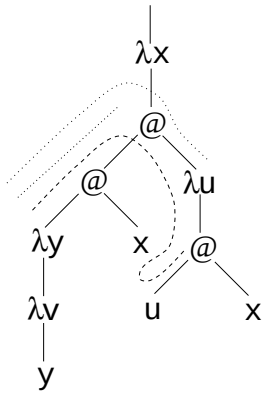


Fig. 5.10. Prerequisite chains.

Exercise 5.5.2 Notice that the definition of prerequisite chain conforms to the idea that variables are back-connected to their binders (as already done for sharing graphs). Then, according to this remark, prove that any prerequisite chain is indeed a path. Moreover, prove that:

(i) Any tail of a prerequisite chain is a prerequisite chain.
(ii) Any prerequisite chain ends with a β-redex (which is indeed the only redex edge crossed by the chain).
(iii) The length of any prerequisite chain is finite.
(iv) If ν is the redex at the end of the prerequisite chain φ of a node $\mathtt{n}$, then:
  (a) the redex ν is to the left of $\mathtt{n}$;
  (b) when $\varphi \neq \nu$, any reduction leading to the creation of a redex involving $\mathtt{n}$ contracts (an instance of) the redex ν.

The previous exercise gives an idea of the information encoded into prerequisite chains. In the proof of next theorem, prerequisite chains will be used to link sharing of Lamping's algorithm to sharing of Lévy's labeling. To this purpose, we need some terminology:

- A *prerequisite chain representation* is a sequence of edges in the sharing graph corresponding to the edges of a prerequisite chain.
- An edge of the syntax tree is called *auxiliary* if does not start at the function port of an application or does not terminate at the context port of an abstraction.

Auxiliary edges are never traversed by prerequisite chains. Neverthe-

less, they are crucial when prerequisite chains are created, as shown by the following example.

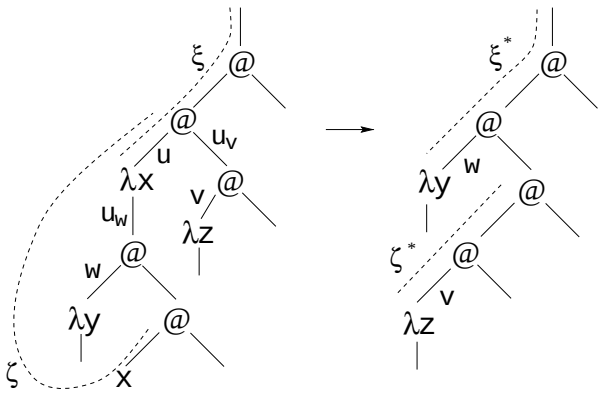


Fig. 5.11. Creation of prerequisite chains.

Example 5.5.3 Let us consider the prerequisite chains $\xi = \xi' \cdot \mathfrak{u}$ and $\zeta = \zeta' \cdot \mathfrak{u}$ in Figure 5.11. After the reduction of $\mathfrak{u}$, two new prerequisite chains are created. Furthermore, these new chains ξ^* and ζ^* respectively involve the auxiliary edges $\mathfrak{u}_w$ and $\mathfrak{u}_v$ (see again Figure 5.11). In fact, with respect to the initial term, they correspond to the lengthening of ξ and ζ by the segments $\mathfrak{u}_w \cdot w$ and $\mathfrak{u}_v \cdot v$ (i.e., we could say that $\xi^* = \xi' \cdot \mathfrak{u} \cdot \mathfrak{u}_w \cdot w$, and that $\zeta^* = \zeta' \cdot \mathfrak{u} \cdot \mathfrak{u}_v \cdot v$).

We are now ready to state and prove optimality of Lamping's algorithm. Let us remark again that what we actually need is the property in the first item of Lemma 5.5.4. Nevertheless, we have to simultaneously prove that also the second item is invariant (i.e., we have to concern about the representation of auxiliary edges), otherwise we would not be able to ensure invariance of the first item under β-reduction.

Before to state the lemma, let us remark that the notion of isomorphism between prerequisite chains is the usual one. Namely, two prerequisite chains are isomorphic when they there is a bijection between their edges and nodes such that labels of the edges are preserved.

Lemma 5.5.4 *Let us assume that the* λ*-term* N *matches with a sharing graph* G *such that* $[M] \twoheadrightarrow G$*, for some term* M *such that* $\mathsf{INIT}(M)$ *is true.*

(i) *Two prerequisite chains in* N *have the same representation in* G *if and only they are isomorphic.*

(ii) *If two auxiliary edges* $\mathtt{n}\,e\,\mathtt{n}'$ *and* $\mathtt{m}\,e'\,\mathtt{m}'$ *have matching labels but different representations, then at least one of the prerequisite chains of* $\mathtt{n}$ *and* $\mathtt{n}'$ *is not isomorphic to the corresponding prerequisite chain of* $\mathtt{m}$ *and* $\mathtt{m}'$.

Proof By induction on the length of the derivation. In the base case (i.e., $[M] = G$), the correspondence between prerequisite chains and representations is injective (any chain has a unique representation). Hence, the theorem trivially holds, for $\mathsf{INIT}(M)$ is true, by hypothesis. Thus, let N' and N be the terms that respectively match the sharing graphs G' and G such that $[M] \twoheadrightarrow G' \xrightarrow{u} G$. We proceed by cases on the rule u.

- If u involves control nodes only or is a permutation between a bracket and a proper node, (i.e., fan-fan, fan-bracket, bracket-bracket, bracket-@, or bracket-λ), then $N = N'$. Also, the rule only causes a local change in the prerequisite chain representations crossing u (a permutation of the nodes involved in the rule, or an erasing in the case of annihilation). Then, by inductive hypothesis we conclude.
- Let u be a fan propagation. The two cases (fan-@ and fan-λ) are similar. So, let us give the case in which u is a fan-@ interaction only. Let @ be the @-node in u. In this case, although we still have $N = N'$, the prerequisite chain representations traversing @ are changed (see the example in Figure 5.9), according to the fact that the fan has propagated through @ duplicating it. Let $@_\circ$ be the copy of @ corresponding to the $\circ$ port of the fan, and $@_*$ the other copy of @. The rule has split the set of prerequisite chain representations crossing @ in two sets (let us note that any prerequisite chain representation in N' crossing @ also crosses the fan in u): the representations in N' traversing the $\circ$ port of the fan are transformed in representations crossing $@_\circ$; the representations in N' traversing the $*$ port of the fan in representations crossing $@_*$. In other words, the rule has just caused some local permutations in prerequisite chain representations, plus the replacement of any occurrence of @ with a corresponding instance $@_\circ$ or $@_*$, according to the way in which the representation traversed the fan. Thus, by induction hypothesis we conclude.
- Let u be a β-interaction. (Let us remark that β-reduction is the unique rule that creates and destroys prerequisite chains; see Example 5.5.3.) By induction hypothesis, the edge u represents

only and all the redexes with a given label. Hence, $\mathcal{U} : N' \twoheadrightarrow N$, where $\mathcal{U}$ is a maximal set of redexes in a given family. Let us prove in order the two properties:

(i) Take two isomorphic prerequisite chains ξ and ζ. The only relevant case is when at least one of them contains an edge created by the reduction of u, otherwise, the property follows immediately by induction hypothesis. Hence, let $\xi = \xi' \cdot \nu \cdot \xi''$, where ν is the first edge in ξ that has been created by u. Correspondingly, we have $\zeta = \zeta' \cdot w \cdot \zeta''$, where ζ' (ζ'') has the same length of ξ' (ξ''), and the edges ν and w yield the same label. By Lemma 5.4.12, also w has been create by the reduction of u. Moreover, both u and w must be of the same type, that is, either they are both result links or they are substitution links. The crucial observation is that ξ and ζ may be seen as compositions of suitable prerequisite chains. More precisely, we have that:

   (a) The edges ν and w may be seen as contractions of two chains $e'_\nu \cdot u_\nu \cdot e''_\nu$ and $e'_w \cdot u_w \cdot e''_w$ with matching labels and such that $u_\nu, u_w \in \mathcal{U}$. In particular, any representation of ν in G is obtained by erasing an @-λ pair from some representation of $e'_\nu \cdot u_\nu \cdot e''_\nu$, and analogously for w and $e'_w \cdot u_w \cdot e''_w$.

   (b) The edges e'_ν and e'_w cannot be auxiliary. Let us prove it for e'_ν. Since ξ is a prerequisite chain, ν cannot be an auxiliary edge. Hence, ν is either the function edge of an application or the context edge of an abstraction. In both cases, we easily see that e'_ν is of the same type of ν, thus, it cannot be auxiliary. Instead, let us note that e''_ν and e''_w can be auxiliary edges.

   (c) Both ξ' and ζ' do not contain edges created by the reduction of any redex in $\mathcal{U}$. Moreover, they are isomorphic, and $\xi' \cdot e'_\nu \cdot u_\nu$ and $\zeta' \cdot e'_w \cdot u_w$ are prerequisite chains in N'. Hence, by induction hypothesis, the representations of $\xi' \cdot e'_\nu \cdot u_\nu$ and $\zeta' \cdot e'_w \cdot u_w$ coincide in G'.

   (d) Both ξ'' and ζ'' are prerequisite chains in the term N (see Exercise 5.5.2). Hence, by induction hypoth-

esis on the number of new edges, the representations of ξ'' and ζ'' in G coincide.

(e) Let us consider the longest subpaths of ξ'' and ζ'' composed of edges that have not been created by the reduction of $\mathcal{U}$ (the first segments of ξ'' and ζ'' obtained applying inductive hypothesis). It is immediate to note that such paths either are prerequisite chains in N' (when they coincide with ξ'' and ζ'') or can be completed into prerequisite chains in N'. Namely, let ξ_0'' and ζ_0'' be the prerequisite chains in N' of the nodes at the end of e_v'' and e_w'', respectively. Such chains are isomorphic. Thus, the representation in G' of ξ_0'' and ζ_0'' coincide.

The previous considerations allows us to conclude that the representation of ξ and ζ in G coincide. In fact, the third and fourth items ensure that the subpaths corresponding to ξ' and ζ' and the subpaths corresponding to ξ'' and ζ'' coincide in any representation of ξ and ζ. Thus, the two representations might differ for the representations of u and v only.

According to the first item in the previous enumeration, the subpaths corresponding to v and w in any representation in G of ξ and ζ correspond to suitable contractions of subpaths representing $e_v' \cdot u_v \cdot e_v''$ and $e_w' \cdot u_w \cdot e_w''$. Furthermore, the third item allows to conclude that the subpaths corresponding to $e_v' u_v$ and $e_w' u_w$ coincide. We left to see what happens to e_v'' and e_w''.

(a) If e_v'' and e_w'' are not auxiliary edges, then the paths $e_v'' \cdot \xi_0''$ and $e_w'' \cdot \zeta_0''$ are prerequisite chains. Since by induction hypothesis they coincide, we conclude that also the subpaths corresponding to v and w in the representations of ξ and ζ coincide.

(b) Let $\mathtt{n}\, e_v'' \,\mathtt{n}'$ and $\mathtt{m}\, e_w'' \,\mathtt{m}'$ be auxiliary edges. We have already proved that the representations of the prerequisite chains of $\mathtt{n}$ and $\mathtt{m}$ (i.e., u_v and u_w) and of the prerequisite chains of $\mathtt{n}'$ and $\mathtt{m}'$ (i.e., ξ_0'' and ζ_0'') coincide. Hence, by the part on auxiliary edges of induction hypothesis, we conclude that also the representation of e_v'' and e_w'' coincide.

The proof of the inverse implication is similar, just start assuming that ξ and ζ have the same representation.

(ii) Let $\mathtt{n}\, e\, \mathtt{n}'$ and $\mathtt{m}\, e'\, \mathtt{m}'$ be two auxiliary edges as in the hypothesis. Furthermore, let us assume that they have not been created by the reduction of $\mathcal{U}$ and, without loss of generality, that the prerequisite chains ξ' and ζ' of $\mathtt{n}$ and $\mathtt{m}$ have different representations in G'. The reduction of $\mathcal{U}$ changes the prerequisite chains of $\mathtt{n}$ and $\mathtt{m}$ only if they end with a redex in $\mathcal{U}$. Anyhow, let ξ and ζ be the new prerequisite chains in N of $\mathtt{n}$ and $\mathtt{m}$, respectively. If $\xi' = \xi$, then the property trivially follows by $\zeta \neq \xi'$: in the case $\zeta = \zeta'$, this inequality holds by hypothesis; in the case that ζ' ends by a redex in u, the prerequisite chain ζ contains at least an edge with a label that where not present in N' (see Lemma 5.4.12). Finally, let us assume that both ξ' and ζ' end with a redex in U. Let $\xi' = \xi'' \cdot e' \cdot u'$ and $\zeta' = \zeta'' \cdot e'' \cdot u''$. Since ξ' and ζ' are not isomorphic by hypothesis, either ξ' is isomorphic to ζ' but e' and e'' are not, or ξ' and ζ' are not isomorphic. In both cases, by Lemma 5.4.12, we see that ξ and ζ must eventually differ after two isomorphic prefixes.

When one of $\mathtt{n}\, e\, \mathtt{n}'$ and $\mathtt{m}\, e'\, \mathtt{m}'$ has been created by the reduction, say $\mathtt{n}\, e\, \mathtt{n}'$, we see that they may have matching labels only if they have are both new (by Lemma 5.4.12). In this case let us note that such new auxiliary edges are the result of the contraction of two sequences $a_0 \cdot u_1 \cdots u_k \cdot a_k$ and $b_0 \cdot v_1 \cdots v_h \cdot b_h$ of auxiliary edges $a_0, \ldots, a_k, b_0, \ldots, b_h$ and redexes $u_1, \ldots, u_k, v_1, \ldots, v_h$ in $\mathcal{U}$. Some thoughts allow to conclude that: $h = k$; the labels of these edges must match; the representations of $\mathtt{n}\, e\, \mathtt{n}'$ and $\mathtt{m}\, e'\, \mathtt{m}'$ may differ only because the representations of a_0 and b_0 or the representations of a_k and b_k differ. Let us take the case in which a_0 and b_0 differ. By induction hypothesis, the prerequisite chains of $\mathtt{n}$ and $\mathtt{m}$ in N' differ. We can then conclude applying the same technique used for the case in which $\mathtt{n}\, e\, \mathtt{n}'$ and $\mathtt{m}\, e'\, \mathtt{m}'$ were not new.

□

The completeness of Lamping's evaluator is an immediate consequence of the above lemma.

Theorem 5.5.5 *For any* λ*-term* M*, we have that* $[M] \twoheadrightarrow G$ *by Lamping's algorithm if and only if* $M \twoheadrightarrow N$ *for some complete (family) reduction and* N *matches with* G*.*

Proof Redexes are particular cases of prerequisite chains. Hence, because of Lemma 5.5.4, all the redexes of a term N' in a same family have a unique representation in the sharing graph G' that N' matches. The proof is then by induction on the length of the derivation.

(*only-if direction*): Since the other rules do not change the matching term, we only consider the case $u : G' \to G$ for some β-redex u. Since, u represent all the redexes in the same family in the term N' matching with G', it corresponds to the complete family contraction $\mathcal{U} : N' \to N$.

(*if direction*): Let us take a maximal set $\mathcal{U}$ of redexes with the same label in N'. All these redexes have the same representation in the matching sharing graph G'. Such a representation φ might not be an edge, for it might contain several control nodes. Nevertheless, there is a reduction sequence composed of control node rules only contracting the path φ to a β-redex. Namely, $G' \twoheadrightarrow G''$ for some G'' that matches N' in which the representation of $\mathcal{U}$ is an edge u. Thus, $G' \twoheadrightarrow G'' \xrightarrow{u} G$ ensures that G matches with the term N such that $\mathcal{U} : N' \to N$.

□

5.6 Optimal derivations

An optimal evaluator must meet two goals: avoiding to duplicate redexes and avoiding to perform useless work. The first goal is satisfied by the completeness requirement; the second one is fulfilled by call-by-need derivations.

The formalization of call-by-need requires some terminology. Let $\rho = \mathcal{F}_1 \cdots \mathcal{F}_n$, the set of redexes in the initial term one of whose residuals is contracted along ρ is defined in the following way:

$$\mathcal{R}(\rho) = \{R \mid \exists i.\ (R/\mathcal{F}_1 \cdots \mathcal{F}_{i-1}) \cap \mathcal{F}_i \neq \varnothing\}.$$

Moreover, as usual, we shall say that a derivation ρ is *terminating* if its final expression is in normal form.

Definition 5.6.1 (call-by-need) A redex R in M is *needed* if and only

if, for every terminating derivation σ starting at M, $R \in \mathcal{R}(\sigma)$. A derivation $\rho = \mathcal{F}_1 \cdots \mathcal{F}_n$ is *call-by-need* if and only if there is at least one needed redex in every $\mathcal{F}_i$.

Proposition 5.6.2 *Every λ-term has at least a needed redex.*

Proof Let R be the leftmost-outermost redex in M and $\rho : M \twoheadrightarrow N$ be a derivation such that $R \notin \mathcal{R}(\rho)$. Then there exists a redex S in N such that $R/\rho = \{S\}$, which means that R is needed. □

Theorem 5.6.3 *Let M have a normal form. Any call-by-need derivation starting at M is eventually terminating.*

Proof By the standardization theorem, the leftmost-outermost derivation of M eventually reaches the normal form. Let $\mathsf{d}(M)$ be the length of the leftmost-outermost derivation starting at M. Let

$$\rho = M \xrightarrow{\mathcal{F}_1} M_1 \cdots M_{n-1} \xrightarrow{\mathcal{F}_n} M_n \cdots$$

be a call-by-need derivation. We prove that

$$\mathsf{d}(M) > \mathsf{d}(M_1) > \cdots > \mathsf{d}(M_n) > \cdots.$$

Let $\sigma = R_1 \cdots R_k$ be the leftmost-outermost derivation reducing M to its normal form. Then $\sigma/\mathcal{F}_1 = \mathcal{C}_1 \cdots \mathcal{C}_k$, where $\mathcal{C}_i = \varnothing$ or $\mathcal{C}_i = \{S_i\}$, since residuals of leftmost-outermost redexes, if any, remain leftmost-outermost. Now, as ρ is a call-by-need derivation, there is a needed redex T in $\mathcal{F}_1$. Hence, $T \in \mathcal{R}(\sigma)$, that is, $R_j \in T/R_1 \cdots R_{j-1}$ for some j. Therefore, $\mathcal{C}_j = \varnothing$ and $\mathsf{d}(M) > \mathsf{d}(M_1)$. Iterating the argument, we get that $\mathsf{d}(M_{i-1}) > \mathsf{d}(M_i)$, for any $i > 1$. □

We can finally define a cost measure for derivations. An evaluator keeping shared copies of a redex performs a reduction of all the copies in one step. Let us give unitary cost to the reduction of a set of copies. According to this, the cost of a reduction $\mathcal{F}$ containing n different sets of copies is n. The cost of a derivation ρ, say $\mathsf{cost}(\rho)$, is the sum of the costs of each reduction in ρ. By Lemma 5.4.8, if ρ is a complete derivation, $\mathsf{cost}(\rho) = |\rho|$. Furthermore, by Proposition 5.4.10, $|\rho| = \sharp(\mathsf{FAM}(\rho))$ for complete derivations ρ. Thus, since for every derivation σ

$$\mathsf{cost}(\sigma) \geq \sharp(\mathsf{FAM}(\sigma))$$

we conclude that complete derivations have minimal cost with respect

to the cost of reducing copies. Nevertheless, in order to be optimal we must avoid useless computations too.

Theorem 5.6.4 (optimality) *Any complete and call-by-need derivation computes the normal form in optimal cost.*

Proof Let ρ be a complete call-by-need derivation and σ be a terminating derivation starting at the same expression as ρ.
Firstly we prove by induction on $|\rho|$ that $\mathsf{FAM}(\rho) \subseteq \mathsf{FAM}(\sigma)$. The base case is obvious. So, let $\rho = \rho'\mathcal{F}$. Then $\mathsf{FAM}(\rho) = \mathsf{FAM}(\rho') \cup \{[\rho' S]\}$, for any $S \in \mathcal{F}$ and, by induction hypothesis, $\mathsf{FAM}(\rho') \subseteq \mathsf{FAM}(\sigma)$. By hypothesis, there exists a needed redex R in $\mathcal{F}$. Then $R \in \mathcal{R}(\sigma/\rho')$, which means that $[\rho' R] \in \mathsf{FAM}(\sigma)$ and $\mathsf{FAM}(\rho) \subseteq \mathsf{FAM}(\sigma)$.
Then, the cost of every terminating reduction σ is greater than the cost of a complete and call-by-need derivation ρ, since

$$\mathsf{cost}(\sigma) \geq \sharp(\mathsf{FAM}(\sigma)) \geq \sharp(\mathsf{FAM}(\rho)) = \mathsf{cost}(\rho)$$

where the last equality follows by Lemma 5.4.8 and Proposition 5.4.10. □

The following corollary is an easy consequence of optimality theorem and completeness of Lamping's algorithm.

Corollary 5.6.5 *Lamping's evaluator equipped with a call-by-need reduction strategy (the leftmost-outermost, for instance) is optimal.*

Nevertheless, let us conclude observing that a call-by-need strategy reduces the possible parallelism of the evaluator. A parallel implementation should instead adopt a strategy different than call-by-need in order to achieve high parallelism, at the cost of performing some useless work.

6
Paths

In the previous chapters, we already remarked that the main difficulties in implementing Lèvy's optimality theory is that it does not suffice to avoid the duplication of reducible subterms; indeed, we must also avoid the duplication of any "virtual redex"—any application which is not yet a redex, for its functional part is not yet an abstraction, but that might become a redex later on during the reduction due to the instantiation of some of its free variables.

A typical example of virtual redex is an application $(y\ z)$ in which the variable y may be instantiated by an abstraction after the execution of a redex. We met a virtual redex like this in the introduction, analyzing the example $M = (\lambda x.(x\ I)\ \lambda y.(\Delta\ (y\ z)))$, where $I = \lambda y.y$ and $\Delta = \lambda x.(x\ x)$ (see Chpater 2). In particular, we observed that the internal strategies are not the shortest ones in reducing M, for the subterm $(y\ z)$ will become a redex $(I\,z)$ after the reduction of the outermost redex of M and of the new redex created by it. Hence, adopting an internal strategy, the redex $(\Delta\ (y\ z))$ would be reduced first, creating two instances of the application $(y\ z)$; the successive instantiation of y with I, would then create two copies of the redex $(I\,z)$, causing a useless duplication of work. At the same time (see Figure 2.2), the analysis of the example pointed out that we could interpret the virtual redex corresponding to $(y\ z)$ as a "path" (in the abstract syntax tree of M) connecting the function port of the application $(y\ z)$ to the root of the term I that will replace y (this path has been redrawn in Figure 6.1). Intuitively, such a "path" describes the "control flow" between the application and the λ-node that gives rise to the virtual redex—note that the root of I is the context port of a λ-node. Furthemore, the path associated to the virtual redex of $(y\ z)$ is the only path that contracts by β-reduction into the edge of the actual redex $(I\,z)$ (see again Figure 2.2 of Chapter 1).

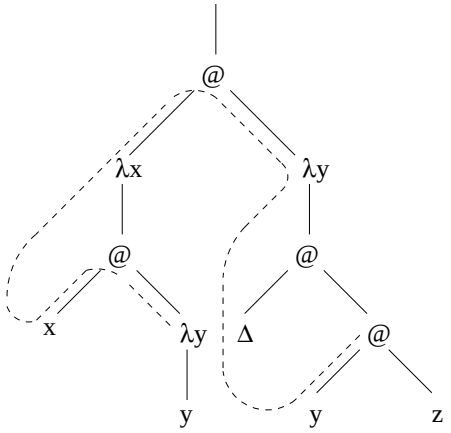


Fig. 6.1. Virtual redexes and paths

Pursuing the analogy with paths, the statement that an optimal implementation has to avoid the duplication of virtual redexes can thus be rephrased by saying that an optimal implementation has to compute the normal form of a term without duplicating (or computing twice) *any path corresponding to a virtual redex.*

In this chapter of the book, we shall investigate the formal definition of these kind of paths, give a formal analysis of their relations with optimality, and present some distinct characterizations of them. But before to go on, let us remind that, even if according to Figure 6.1 the "path" of the virtual redex $(\mathsf{y}\ \mathsf{z})$ is not an actual path in the usual sense, it turns out to be an actual path as soon as we assume that bound variables are explicitly connected to their binders—as in the sharing graph notation we are using.

6.1 Several definitions of paths

As a matter of fact, the study of paths definable on the syntax tree representation of λ-terms arosed in the literature from three different perspectives, respectively related to three major foundational investigations of β-reduction: 1) Lévy's analysis of families of redexes (and the associated concept of labeled reduction); 2) Lamping's graph-reduction algorithm; 3) Girard's geometry of interaction. All the previous studies happened to make crucial (even if not always explicit) use of a notion of path, respectively called *legal*, *consistent* and *regular*.

We are already familiar with the notion of Lévy's redex family. Lévy's idea was to use labels to capture the notion of two redexes being created in the "same" way during a reduction. To this purpose, he labeled terms and made beta reductions act on such labels. He gave then a calculus in which two redexes are in the same family if and only if their labels coincide. The intuition that led Lévy to this result was that labels should be a trace of the history of creation of the redex. In [AL93b], Asperti and Laneve made such an intuition more formal, proving that the notion of "history" encoded by labels has a direct image as a "path" in the graph of the initial term. In a certain sense, they showed that the history of a redex can be seen as a sort of flow of control between a suitable application and a suitable λ-node of the term—that is, a virtual redex—and that such flow can be described as a suitable path, say a *legal path*, connecting an application and an abstraction node. The relevant point of legal paths is that their definition is independent from λ-term reduction. In fact, redexes are the base case of legal paths; the inductive construction consists then in lengthening previously built paths according to a simple and effective condition asking for the minimal symmetry necessary to ensure that any legal path might unfold into a redex. Asperti and Laneve showed then the relations between the statical notion of legal path and β-reduction proving that: 1) labels of redexes in any reduct N of M denote paths in M; 2) all the latter paths are legal; 3) any legal path in M denotes a label of a redex to appear somewhere in the set of reducts of M.

Meanwhile, people were seeking for a shared reduction faithfully implementing the notion of families, i.e., a reduction where families could be said to be reduced in one step. The result was the optimal algorithm that we have widely presented in Chapter 3. Also this algorithm rests on a notion of path. In fact, proving its correctness we have already met the so-called *proper paths*, by which we gave the correspondence between sharing graphs and λ-terms. In this chapter, we will present and use instead another notable notion of paths related to the optimal algorithm, the so-called *consistent paths* introduced by Gonthier, Abadi and Lévy in [GAL92a]. In some sense, proper paths are a restriction of consistent paths to the paths of a sharing graph with a direct correspondence in the syntax tree of the matching λ-term.

Finally, Girard unveiled in [Gir89a, Gir88] a more mathematical interpretation of the cut-elimination procedure of linear logic. Again, this alternative computation could be defined as the computation of a particular set of paths defined on the nets representing linear logic proofs.

Namely, as the computation and composition of a set of *regular paths* defined through an algebraic and computational device, the so-called *dynamic algebra* (see the work of Danos and Regnier [Dan90, Reg92, DR95a], and in particular [DR93], where such an approach is extended to pure lambda-calculus).

The surprising fact is that, although the previous three notions above seems to bear no clear relation with each other, they are indeed *equivalent*.

6.2 Legal Paths

The notion of *legal path* arises from the attempt to provide a formal characterization of the intuitive notion of "virtual redex"—as we have already seen, a configuration inside a term which is not yet a redex, but that might become a redex along the reduction of the term (more precisely, there is at least a reduction of the term that transforms such an application into a redex). Let us consider for instance the (labeled) λ-term $(\Delta\,\Delta)$, represented in Figure 6.2.

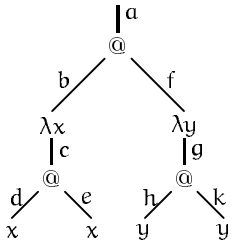


Fig. 6.2. $(\Delta\,\Delta)$

Let us take the application node labeled by c (note that the label of the edge above a node is the label of the corresponding subterm in the labeled calculus). Our question is: "is it possible that this application node might be ever involved in a β-reduction?". In order to answer to such a question, we must look for a λ-node to match against the application. We start our search towards the left son of the @-node. (According to Lafont's terminology [Laf90, AL93a], this is the *principal port* of the application, that is, the only port where we may have interaction with a dual operator; see the definition of β-rule.) Thus, we pass the edge labeled by d and we find a variable. If the variable had been free, we

would have finished: no redex involving our application could possibly exist; but this is not the case. Besides, since the variable is bound, the "control" can be passed back to its binder, that is the λ-abstraction labeled by b in the picture. In fact, a β-reduction involving such a binder would replace the variable with the argument of the initial application. So, to continue our search we have to enter inside such an eventual argument. Namely, we have to pose a symmetrical question about the λ-node we have reached: "is it possible that it would be ever involved in a β-reduction?". Since the latter question concerns a λ-node, we must now travel towards the root (note that this is still the principal port of the λ-node, according to Lafont). We find an application. Moreover, for we enter the application at its principal port, we have got a redex. The second question is then positively solved; so, we can resume our initial question, continuing our search looking into the argument of the application (that is, the argument that would replace the occurrence of y after firing the redex b). Moving from the argument port of the application we cross the edge f and we eventually reach the principal port of a λ node, that is, we finally find a (virtual) redex for the @-node from which we started our search.

As a result of the previous process, in addition to a positive answer to our starting question, we have also collected an ordered sequence of edges dbf. Moreover, assuming that bound variables are explicitly connected to (positive, auxiliary ports of) their respective binders—an idea going back to Bourbaki that we have already used for sharing graphs; such a sequence is indeed connected, that is, it is a "path" in the usual sense. Because of this, in the following, even if we shall not explicitly draw such connections in the pictures, we will always assume that, as for sharing graphs, also in the syntax trees of λ-terms variables are back-connected to their binders.

Summarizing, we may say that the path dbf represents the "virtual redex" (y y). In fact, by firing the redex b, the path dbf "contracts" into a single edge corresponding to the "actual" redex associated to the path dbf.

Let us now consider another example using the same λ-term. Starting at the @-node whose top edge is labeled by g, we travel along the edge h (that is, back to the binder of y) and then up along f. As in the former example, we find an @-node, but in this case entering it at argument port (its negative auxiliary port, not its principal port!). So, we do not find a redex, and to continue we must open a new session, looking for a (virtual) redex involving the @-node whose context edge is labeled by

a. As the function edge b of the @-node is a redex, this new session immediately ends with success, and to continue we have to resume the initial search. But, since at the end of the inner session the "control" was coming from the argument of the @-node labeled by a, we must pass control to some of the variables bound by the λ-node in the redex b, that is d or e (note the non-determinism of the search algorithm at this point). Suppose to follow the first possibility. We find the same redex dbf of the previous computation (even if traveling in the opposite direction). So, we may resume the initial session proceeding through the edge e, then, moving from the binder of x, up along the redex b, and finally down along f. We get in this way the path hfbdebf, beginning at the principal port of the @-node (y y) and ending at the principal port of the λ-node λy. Also in this case, the path we get corresponds to a virtual redex, the one that is image of the unique redex that would be created after two reductions of (Δ Δ) (see also Example 6.2.4). Furthermore, by firing first the redex b and then the (newly created) redex dbf, the path hfbdebf contracts into a single redex-edge.

The previous two examples give an informal description of the search algorithm by which to find virtual redexes, or more precisely, by which to build path descriptions of virtual redexes. Unfortunately, the previous examples are greatly simpler than the general case, so the search algorithm is indeed more contrived than one could expect analyzing the case met so far. Besides, the second path that we have built presents already the critical issue of the general case, for at a certain point we add to choice between two occurrences of the same variable. In the example, we said that such a choice introduces in the algorithm a certain degree of non-determinism, for in this case both were valid choice; on the contrary, in certain situations, it is not true that coming down through a λ the algorithm can proceed non-deterministically, for in such cases the choice of the occurrence of the bound variable must indeed accord with the part of the path already built to reach the λ-node. A good explanation of the problem can be found by inspection of the following example.

Example 6.2.1 Consider the λ-term (λx. ((x M) (x N)) λy. y), represented in Figure 6.3. Let us suppose we are looking for a λ-node to match against the application labeled c. Moving from this application, we go down through d, down again through f, back to b, and then down through l. We find in this way a λ-node matching the application labeled d. Since we entered this application from the top, we continue

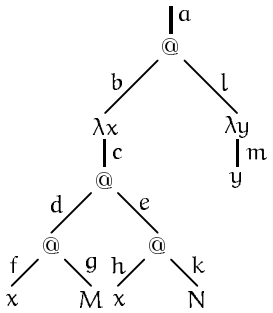


Fig. 6.3. $(\lambda x.((x\,M)\,(x\,N))\,\lambda y.y)$

our search inside the body of the λ. So, we go down through m, back through l, and down through b. We reach in this way the principal port of the λ-node λx, and to continue we have to choose an occurrence of its variable x. In the previous examples, such a choice was free, in this case instead, only the choice of the occurrence labeled f is correct. The intuition should be clear: since we entered the argument of the application coming from this occurrence of the variable, we are conceptually working inside the instance of the argument replacing such a particular occurrence x (in some reduction of the initial term); thus, we cannot jump to a different instance, it would be as jumping from a point of a subterm to a corresponding point of a distinct instance of the same subterm. Because of this, let us assume we correctly choose f: we find (again, but traveling in the opposite direction) the application matching the λ labeled l. Since we enter this λ through the bound variable, we must exit the application through the argument port labeled g, pursuing the search inside M. If M starts with a λ, we have found the virtual redex we were looking for; otherwise, there is no virtual redex between the application labeled c and M. On the contrary, because of the previous arguments about the choice of the occurrence of x to pass through after the path dfblmlb, no virtual redex can exist between the application labeled c and N, independently from the shape of N.

The previous example points out that the involved case arises in presence of a loop inside the argument P of an application. In such a situation we are in fact forced to follow the same path we used to enter the argument P (even if crossing it in the opposite direction), up to the occurrence of the variable indicating the particular instance of P we were

considering. The precise formalization of the previous intuition requires a complex inductive definition that will be discussed in section 6.2. Our aim, for the moment, was just to provide the main intuitions behind the path-description of "virtual" redexes in a λ-term T. According to such an intuition, every "virtual" redex of T defines a suitable "legal" path in the syntax tree of T connecting the function part of an application to the top of a λ-node. The natural question at this point is if also the converse hold, that is, do different "virtual" redexes define different paths? Due to a duplication issue, the answer to this question is instead no. For instance, let us consider a term $T = (\lambda x.M\ N)$ with a redex r inside N. The redex r may have several residuals in $M[N/x]$; even though all these different residuals correspond to the same path in T. This fact, that at a first glance might seem negative, suggests instead that we can reasonably hope to have an injection from "virtual" redexes to paths, up to some notion of "sharing" that we should hope to be the optimal one of Lévy. By the way, this is indeed the case, for there is a *bijective correspondence* between Lévy's families of redexes and the class of legal paths that we have informally introduced by the previous examples. In other words:

two redexes are in a same Lèvy's family, if and only if their associated paths in the initial term of the derivation coincide.

The relevant point with this correspondence is that the formal definition of legal path (see section 6.2) get rid of the labels assigned to the edges of the (labeled) λ-term. Thus, the notion of *legal path* provides a complete characterization of virtual redexes independent from labels.

6.2.1 Reminder on labeled λ-calculus

For ease of reference, we recall the main definitions of the labeled λ-calculus.

Let $L = \{a, b, \ldots\}$ be a denumerable set of *atomic labels*. The set $\mathbf{L}$ of *labels*, ranged over by $\ell, \ell', \ldots, \ell_0, \ell_1, \ldots$, is defined by the following rules:

$$a \quad | \quad \ell_1\ell_2 \quad | \quad \underline{\ell} \quad | \quad \overline{\ell}$$

The operation of concatenation of labels $\ell_1\ell_2$ is assumed to be associative.

The set $\mathbf{L}_p$ of *proper labels* contains exactly all those labels $\ell \in \mathbf{L}$ such that ℓ is atomic or $\ell = \underline{\ell'}$ or $\ell = \overline{\ell'}$.

The set $\mathbf{L}_p$ will be ranged over by $\alpha, \beta, \ldots$.

Given a label ℓ, $\mathsf{at}(\ell)$ is the set of all its *atomic* (sub)labels.

The set $\Lambda_V^{\mathbf{L}}$ of labeled λ-terms over a set V of variables and a set $\mathbf{L}$ of labels is defined as the smallest set containing:

(i) x^ℓ, for any $x \in V$ and $\ell \in \mathbf{L}$;
(ii) $(M\ N)^\ell$, for any $M, N \in \Lambda_V^{\mathbf{L}}$ and $\ell \in \mathbf{L}$;
(iii) $(\lambda x.\, M)^\ell$, for any $M \in \Lambda_V^{\mathbf{L}}$ and $\ell \in \mathbf{L}$.

Labeled β-reduction is the following rule (remind that $\ell_0 \cdot (T)^{\ell_1} = (T)^{\ell_0 \ell_1}$, where T is either a variable, a composition of labeled terms, or an abstraction of a labeled term):

$$((\lambda x.M)^{\ell_0}\ N)^{\ell_1} \to \ell_1 \cdot \overline{\ell_0} \cdot M[\underline{\ell_0} \cdot N/x]$$

The label ℓ_0 is the *degree* of the redex $((\lambda x.M)^{\ell_0}\ N)^{\ell_1}$.

Let M be a labeled λ-term. The predicate $\mathsf{INIT}(M)$ is true if and only if the labels of all subterms of M are atomic and pairwise distinct.

Given a labeled term M, we shall denote by M_I a labeled λ-term equal to M up to labeling, such that $\mathsf{INIT}(N^I)$ is true. If $M \overset{\sigma}{\twoheadrightarrow} N$ and u is a redex of M, we shall denote by $\mathsf{degree}_M^\sigma(u)$ the degree of u after the labeled reduction of M^I isomorphic to σ; at the same, we will continue to use σ to denote such an isomorphic reduction, i.e., $M^I \overset{\sigma}{\twoheadrightarrow} N^\star$, where $N^\star$ is equal to N up to sharing. In other words, two labeled reductions are equated when they correspond to the same unlabeled one.

The map $(\,.\,)^I$ is a sort of *standard initial labeling* function associating a labeled term M^I to each (labeled) term M, with the proviso that $\mathsf{INIT}(M^I)$ holds for any M. According to this, $\mathsf{degree}_M^\sigma(u)$ is the degree of the redex u with respect to the reduction σ and to the standard initial labeling of M

The set $\mathbf{L}_0(M)$ is the set of all the labels contained in the labeled term M^I; the set $\mathbf{L}(M)$ is the set of all the labels obtainable reducing M^I, that is, $\mathbf{L}(M) = \bigcup\{\mathbf{L}_0(N) \mid M^I \twoheadrightarrow N\}$.

Let us remark that both the definition of degree and $\mathbf{L}$ are independent from the actual initial labeling of the term M. In fact, in both of them we use its standard initial labeling. This, in conjunction with the assumption that we equates reductions up to the labels of the terms involved, will allow us to freely mix labeled and unlabeled reductions. In fact, when not otherwise specified, the labeled reductions we are interested in is the one in which the labels of the initial term are assigned according to the standard initial labeling.

6.2.2 Labels and paths

Labels provide a very simple approach to the notion of computation as a travel along a path. In particular, every label trivially defines a path in the initial term. The interesting problem will be to provide an independent characterization of these paths (see Section 6.2).

Before to see the correspondence between labels and paths, let us remind that in the graph representation of λ-terms that we assumed to use, bound variables are explicitly connected to the respective binders—this assumption is obviously crucial in order to obtain connected paths. Furthermore, let us also note that, given a labeled term M, every edge of λ-term $M^{\mathtt{I}}$ is labeled with a distinct atomic symbol; as a consequence, when it will not cause confusion, each edge of M will be denoted by the the label marking it in $M^{\mathtt{I}}$.

Definition 6.2.2 If ℓ is the label of an edge generated along some reduction of the labeled λ-term $M^{\mathtt{I}}$ (i.e., $\ell \in \mathbf{L}(M)$), the *path* of ℓ in the λ-term M is inductively defined as follows:

$$\begin{aligned} \mathsf{path}(\mathfrak{a}) &= \mathfrak{a} \\ \mathsf{path}(\ell_1\ell_2) &= \mathsf{path}(\ell_1)\cdot\mathsf{path}(\ell_2) \\ \mathsf{path}(\overline{\ell}) &= \mathsf{path}(\ell) \\ \mathsf{path}(\underline{\ell}) &= (\mathsf{path}(\ell))^{\mathtt{r}} \end{aligned}$$

where $\mathsf{path}(\ell_1)\cdot\mathsf{path}(\ell_2)$ denotes the concatenation of the corresponding two paths (in the following we will usually omit "$\cdot$" concatenating paths), while $(\mathsf{path}(\ell))^{\mathtt{r}}$ is the path obtained reversing $\mathsf{path}(\ell)$.

By induction on the length of the derivation generating the label ℓ, it is readily seen that the definition of path is sound, i.e., that $\mathsf{path}(\ell)$ is a path of M (it is indeed an immediate consequence of Lemma 6.2.5). Furthermore, $\mathsf{path}(\ell)$ connects nodes/ports of the same type of the ones connected by the edge labeled by ℓ.

Fact 6.2.3 *If ℓ is the label of an edge $\mathfrak{u}$ generated along some reduction of $M^{\mathtt{I}}$, then $\mathfrak{u}$ and $\mathsf{path}(\ell)$ connects corresponding ports of nodes of corresponding type. For instance, if $\mathfrak{u}$ is a redex, then $\mathsf{path}(\ell)$ connects the function (left) port of an application of M to the context (top) port of an abstraction of M.*

The intuitive idea behind the definition of the paths associated to la-

bels is that a redex with degree ℓ is eventually obtained by "contraction" of $\mathsf{path}(\ell)$.

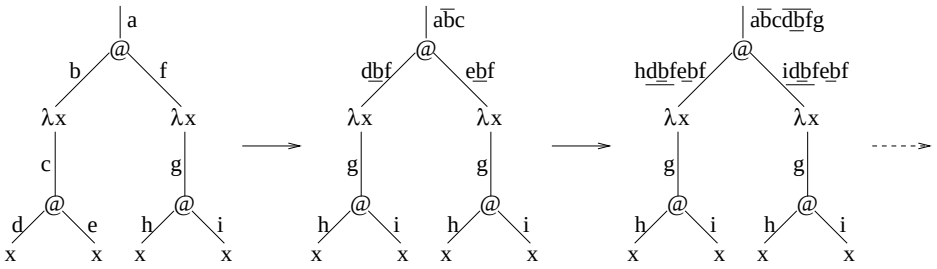


Fig. 6.4. $(\lambda x.(x\,x)\;\lambda x.(x\,x))$

Example 6.2.4 Consider again the term $(\Delta\,\Delta)$ given at the beginning of the chapter. After two reductions, we obtain a redex with degree $\ell = \mathtt{h\underline{d}\underline{b}\underline{f}\underline{e}\underline{b}f}$ (see Figure 6.4). Thus, $\mathsf{path}(\ell) = \mathtt{hfbdebf}$, that is the path we obtained in the introduction to the chapter for the application labeled with $\mathtt{g}$.

The interesting point is that different degrees define different paths in the initial term. The proof of such a claim requires a better comprehension of the structure of labels. Hence, as a preliminary remark, let us note that any label ℓ can be written as the concatenation of a (unique) sequence of proper labels $\alpha_1 \cdots \alpha_n$. Then, let us analyze in more detail the shape of such a sequence of proper labels.

Lemma 6.2.5 *Let $\alpha_1 \cdots \alpha_n \in \mathbf{L}(M)$; where $\alpha_i \in \mathbf{L}_p$ (i.e., α_i is either atomic or an over/underlined label), for $i = 1, \ldots, n$. Then:*

(i) *The length n of $\alpha_1 \cdots \alpha_n$ is odd.*

(ii) *For every i odd, α_i is atomic.*

(iii) *For every i even (i.e., $i = 2j$), either $\alpha_i = \overline{\ell}$ or $\alpha_i = \underline{\ell}$, for some $\ell \in \mathbf{L}(M)$ such that $\mathsf{path}(\ell)$ starts at the function (left) port of an application and ends at the context of an abstraction of M. Furthermore:*

(a) *when $\alpha_{2j} = \overline{\ell}$, then:*

1. *α_{2j-1} is the context (top) edge of the @-node from which $\mathsf{path}(\ell)$ starts;*
2. *α_{2j+1} is the context (top) edge of the λ-node to which $\mathsf{path}(\ell)$ ends;*

(b) *when* $\alpha_{2j} = \underline{\ell}$, *then:*

1. α_{2j-1} *is a binding edge of the* λ*-node to which* $\mathsf{path}(\ell)$ *ends;*
2. α_{2j+1} *is the argument (right) edge of the* @*-node from which* $\mathsf{path}(\ell)$ *starts.*

Proof By an easy induction on the length of the derivation. □

Remark 6.2.6 An immediate corollary of the previous proposition is that the degree ℓ of a redex may appear inside another label only over/underlined and surrounded by atomic labels. Vice versa, any over/underlined label is the degree of some redex.

Another consequence of the above proposition is that both over and underlines can be erased from Lévy's labels without any loss of information, as the lining of such a flat label can be recovered using the result of the previous lemma.

Corollary 6.2.7 *Overlining and underlining could be safely omitted in the definition of labeled* λ*-calculus, i.e., its* β*-rule could be replaced by a rule equivalent in all respects to the old one but for the fact that the lines above or below labels are omitted.*

Proof Remove all the over/underlines contained in a label ℓ. This can be formalized by a function flat equal to path whose definition is similar to the one of path except for $\mathsf{flat}(\underline{\ell}) = \mathsf{flat}(\ell)$. The initial structure of ℓ can be inductively retrieved applying Lemma 6.2.5 and working inside-out. Namely, starting from the atomic labels of $\mathsf{flat}(\ell)$ relative to redexes, we over/underline each of them them according to the labels surrounding it (note that this step is uniquely determined by the initial labeling of the term). Iterating the process with the structured labels yielded so far, we eventually get a label ℓ' such that $\mathsf{flat}(\ell') = \mathsf{flat}(\ell)$. Since the procedure applied to build ℓ' is deterministic, we conclude that ℓ' is the unique label having this property that accords to the properties of Lemma 6.2.5. That is, $\ell' = \ell$. □

The deep reason behind the previous result is the tight correspondence between paths and labels that we are pursuing. With this respect the first property we can show is that two distinct labels arising during the reduction of a term $M^{\mathtt{I}}$ always correspond to distinct paths. The relevant point is that in this claim we do not make any assumption on

the terms in which the two labels appear, so they might also mark edges of two distinct terms.

Proposition 6.2.8 *The function* path *is injective over* $\mathbf{L}(\mathsf{M})$.

Proof Let ℓ_1 and ℓ_2 be two distinct labels in $\mathbf{L}(\mathsf{M})$. The proof that $\mathsf{path}(\ell_1) \neq \mathsf{path}(\ell_2)$ is by induction on the structure of ℓ_1. The case ℓ_1 atomic is immediate. Thus, let $\ell_1 = \alpha_1 \cdots \alpha_n$. If $\ell_1 = \ell_2\ell$ (or vice versa), the thesis follows trivially, since the two paths have different lengths. Anyhow, let us note that the previous situation is not the case when either ℓ_1 or ℓ_2 is the degree of some redex (by the remark on Lemma 6.2.5). Hence, let us assume that $\ell_2 = \beta_1 \cdots \beta_m$ and that k is the first index such that $\alpha_k \neq \beta_k$. According to Lemma 6.2.5, we can distinguish two cases:

(k *is odd*) In this case, both α_k and β_k are atomic, that is, α_k and β_k are distinct edges of M. Thus, the two paths diverge here.

(k *is even*) In this case, either $\alpha_k = \underline{\ell_1'}$ or $\alpha_k = \overline{\ell_1'}$, for some degree ℓ_1'; and analogously for β_k and some degree ℓ_2'. Furthermore, either both α_k and β_k are overlined or both of them are underlined (i.e., it is not the case that $\alpha_k = \underline{\ell_1'}$ and $\beta_k = \overline{\ell_2'}$, or vice versa), for by hypothesis $\alpha_{k-1} = \beta_{k-1}$. Since $\ell_1', \ell_2' \in \mathbf{L}(t)$ and $\ell_1' \neq \ell_2'$, by inductive hypothesis we get that $\mathsf{path}(\ell_1') \neq \mathsf{path}(\ell_2')$. As we already remarked, in such a case neither ℓ_1' can be a prefix of ℓ_2', nor ℓ_2' can be a prefix of ℓ_1'. So, since both $\mathsf{path}(\ell_1')$ and $\mathsf{path}(\ell_2')$ start at the function port of the same application and end at the context port of the same abstraction, there are two paths ζ_1 and ζ_2 starting and ending with distinct edges such that $\mathsf{path}(\ell_i') = \phi\zeta_i\psi$, for two non empty paths ϕ and ψ and $i = 1, 2$. We conclude then that $\mathsf{path}(\ell_1)$ and $\mathsf{path}(\ell_2)$ diverge after a common prefix $\xi\psi^r$ or $\xi\phi$, where $\xi = \mathsf{path}(\alpha_1 \cdots \alpha_{k-1})$. In fact, when α_k and β_k are underlined, we get $\mathsf{path}(\ell_1) = \xi\psi^r\zeta_1^r\phi^r\xi_1$ and $\mathsf{path}(\ell_2) = \xi\psi^r\zeta_2^r\phi^r\xi_2$, for some paths ξ_1 and ξ_2; while when α_k and β_k are overlined, we get $\mathsf{path}(\ell_1) = \xi\phi\zeta_1\psi\xi_1$ and $\mathsf{path}(\ell_2) = \xi\phi\zeta_2\psi\xi_2$.

□

Exercise 6.2.9 Give an algorithm defining a function `pathtolab` from paths of M to Lévy's labels such that $\mathtt{pathtolab}(\mathsf{path}(\ell)) = \ell$, for any $\ell \in \mathbf{L}(\mathsf{M})$.

6.2.3 The equivalence between extraction and labeling

In Chapter 5 we have shown that all the redexes of a family bear the same degree (see Theorem 5.3.41). To complete the proof of the equivalence between extraction and labeling, we need to prove that two redexes have the same degree only if they are in the same family. (Let us remind again that a key assumption in this is that INIT holds for the initial term; but, as all the labels we will compute are relative to the reduction of a term $M^{\mathtt{I}}$, this is definitely the case.)

The correspondence between labels and paths is expressed by the following theorem, of which we have to prove the "if" part only, for the "only if" directions is an immediate consequence of Theorem 5.3.41.

Theorem 6.2.10 *Let* $M \stackrel{\rho}{\twoheadrightarrow} N_1$ *and* $M \stackrel{\sigma}{\twoheadrightarrow} N_2$. *Then* $\rho u \simeq \sigma v$ *if and only if* $\mathsf{degree}^{\rho}_{M}(u) = \mathsf{degree}^{\sigma}_{M}(v)$.

The correspondence between paths and labels we presented in the first part of this chapter gives a way to complete the proof of this equivalence. In order to give the full details of such a proof, we need the following two preliminary lemmas.

Lemma 6.2.11 (First Redex Lemma) *Let* σv *be a redex with history* σ *such that* σv *is canonical. Let* u *be the leftmost-outermost redex of* M *for which* $\mathsf{degree}^{\epsilon}_{M}(u) \in \mathsf{at}(\mathsf{degree}^{\sigma}_{M}(v))$. *Then* u *is the first redex fired by* σ.

Lemma 6.2.12 (Contracted Label Lemma) *Let* $M \stackrel{w}{\to} N \stackrel{\rho}{\twoheadrightarrow} N_1 \stackrel{u}{\to}$ *and* $M \stackrel{w}{\to} N \stackrel{\sigma}{\twoheadrightarrow} N_2 \stackrel{v}{\to}$ *be two derivations such that both* $w\rho u$ *and* $w\sigma u$ *are canonical. Then*

$$\mathsf{degree}^{w\rho}_{M}(u) \;=\; \mathsf{degree}^{w\sigma}_{M}(v) \quad \Rightarrow \quad \mathsf{degree}^{\rho}_{N}(u) \;=\; \mathsf{degree}^{\sigma}_{N}(v).$$

We postpone the proof of such lemmas to the end of the section, after the proof of the main theorem. The intuitive idea behind such proofs is very simple. In the case of the first redex lemma, we observe that every redex traversed by the path associated to the degree of a redex v is needed for its creation. As a consequence, the leftmost-outermost of such redexes should definitely be the first one fired along the derivation σ such that σv is canonical. In the case of the second lemma, we already know that if $\mathsf{degree}^{\rho}_{N}(u) \;\neq\; \mathsf{degree}^{\sigma}_{N}(v)$ they define different paths in N. Now, in any labeled reduction starting with $M^{\mathtt{I}}$, the only possibility that these paths get a same labeling is that they are isomorphic paths

inside different instances of the argument of the redex w. But this case is excluded by canonicity, for in such a case we could apply the last rule of the extraction relation (see Definition 5.2.2).

6.2.3.1 Proof of Theorem 6.2.10

The "only if" direction of the theorem has been already proved by Theorem 5.3.41. Thus, we left to show that:

$$\mathsf{degree}^{\rho}_{M}(u) = \mathsf{degree}^{\sigma}_{M}(v) \quad \Rightarrow \quad \rho u \simeq \sigma v$$

As a consequence of the fact that $\rho u \simeq \sigma v \Rightarrow \mathsf{degree}^{\rho}_{M}(u) = \mathsf{degree}^{\sigma}_{M}(v)$ has been already proved, we can restrict to the case in which ρu and σv are canonical. Furthermore, because of the uniqueness of the canonical form, this also means that the former implication is equivalent to:

$$\rho u \neq \sigma v \quad \Rightarrow \quad \mathsf{degree}^{\rho}_{M}(u) \neq \mathsf{degree}^{\sigma}_{M}(v)$$

where ρu and σv are canonical.

The proof proceed by induction on the length of ρ:

(*base case*: $\rho = \epsilon$) In this case u is a redex of the initial term M; its label is thus atomic. As a consequence, the property immediately holds when $\mathsf{degree}^{\sigma}_{M}(v)$ is not atomic. Furthermore, let us assume that $\mathsf{degree}^{\sigma}_{M}(v)$ is atomic. Since σv is canonical, $\sigma = \epsilon$ (we invite the reader to check that, if this was not the case, then we could apply at least one of the rules of the extraction relation). Then, by the definition of degree, we conclude noting that $\mathsf{degree}^{\epsilon}_{M}(u) \neq \mathsf{degree}^{\epsilon}_{M}(v)$ if and only if $u \neq v$.

(*induction case*: $\rho = w\rho'$) Let $\sigma = w'\sigma'$ (let us assume without loss of generality that ρ is not longer than σ). By hypothesis, ρ and σ are standard; then, by Lemma 6.2.11, $\mathsf{degree}^{\epsilon}_{M}(w)$ and $\mathsf{degree}^{\epsilon}_{M}(w')$ are the labels of the leftmost-outermost redexes in $\mathsf{at}(\mathsf{degree}^{\rho}_{M}(u))$ and $\mathsf{at}(\mathsf{degree}^{\sigma}_{M}(v))$, respectively. As an immediate consequence, $\mathsf{degree}^{\rho}_{M}(u) \neq \mathsf{degree}^{\sigma}_{M}(v)$, when $w \neq w'$. Hence, let us assume $w = w'$ and $M \xrightarrow{w} N$. In such a case $\rho u \neq \sigma v$ only if $\rho' u \neq \sigma' v$. Then, by the inductive hypothesis, we get $\mathsf{degree}^{\rho}_{N}(u) \neq \mathsf{degree}^{\sigma}_{N}(v)$, and by Lemma 6.2.12, we conclude that $\mathsf{degree}^{w\rho}_{M}(u) \neq \mathsf{degree}^{w\sigma}_{M}(v)$.

The original proof given by Lévy in his thesis ([Lév78, pp. 68–113]) was much contrived than the one presented here. Lévy's proof based on a complex notion of labeled subcontexts, and it takes over than forty

pages. It is interesting to note that his induction on the length of the derivation worked in the opposite direction, i.e., considering the tail redex of the canonical derivations. The main reason behind the simplification of the proof presented here (that was published for the first time in [AL93b]) is that it takes advantage of the structural properties of labels that Lévy did not notice at that time—in particular, it rests on the correspondence between labels and paths.

6.2.3.2 Proof of the First Redex Lemma (Lemma 6.2.11)

By induction on the length of the derivation σ. The base case ($\sigma = \epsilon$) is vacuous. Hence, let $\sigma = u\sigma'$ and $M^{\mathrm{I}} \xrightarrow{u} N$. We proceed then by induction on σ':

($\sigma' = \epsilon$) Since uv is canonical, v is not a redex of M, it has instead been created by executing u. In more details, let v be $((\lambda y.\, S)^{\ell}\ T)^{\ell'}$ in N. If the redex u is $((\lambda x.\, P)^{a}\ Q)^{b}$ in M^{I} (note that, since we are considering M^{I}, the labels in u are atomic), then $((\lambda x.\, P)^{a}\ Q)^{b} \xrightarrow{u} b \cdot \underline{a} \cdot P[\overline{a} \cdot Q/x]$. Accordingly, we have two possibilities: (*i*) the term M^{I} contains $(((\lambda x.\, P)^{a}\ Q)^{b}\ T)^{\ell'}$ and $P = (\lambda y.\, S')^{c}$, with $S = S'[\overline{a} \cdot Q/x]$; (*ii*) $Q = (\lambda y.\, S)^{c_1}$ and the term P contains $((x)^{c_2}\ T')^{\ell'}$, where x is bound by the abstraction of u and $T = T'[\overline{a} \cdot Q/x]$. It is immediate to see that, correspondingly, we have either $\ell = b\underline{a}c$ and $\ell = c_2\overline{a}c_1$, that is, a is the unique degree of a redex contained in $\mathsf{at}(\ell)$. Hence, as $\ell = \mathsf{degree}^{\sigma}_{M}(v)$ and $a = \mathsf{degree}^{\epsilon}_{M}(u)$, we conclude.

($\sigma' = w\sigma''$) Let $\mathsf{degree}^{\epsilon}_{M}(u) = a$. For any degree ℓ of a redex of N, only two cases may hold: (*i*) ℓ is atomic and $\ell \in \mathbf{L}_0(M)$; (*ii*) there are two atomic labels $b, c \in \mathbf{L}_0(M)$ such that $\ell = b\alpha c$, with $\alpha \in \{\underline{a}.\overline{a}\}$. Furthermore, in the first case, the redex is the residual of a redex of M; in the second, the redex has been created by the execution of u, and none of the atomic labels b and c can be the degree of a redex of M. As an immediate consequence, we get then that $\mathsf{at}(\mathsf{degree}^{\sigma}_{M}(v)) = \mathsf{at}(\mathsf{degree}^{\sigma'}_{N}(v)) \cup a$ (we invite the reader to prove it using the results of Exercise 6.2.13). Hence, for any redex $u' \neq u$ such that $\mathsf{degree}^{\epsilon}_{M}(u') \in \mathsf{at}(\mathsf{degree}^{\sigma}_{M}(v))$ and any redex $w' \in u'/u$, then $\mathsf{degree}^{\epsilon}_{N}(w') \in \mathsf{at}(\mathsf{degree}^{\sigma'}_{N}(v))$. By the induction hypothesis (note that $\sigma' v = w\sigma'' v$ is canonical), w is the leftmost-outermost redex of N for which $\mathsf{degree}^{\epsilon}_{N}(w) \in \mathsf{at}(\mathsf{degree}^{\sigma'}_{N}(v))$. So, w is leftmost-outermost with respect to w'. By this and the hypothesis that σ is canonical, it follows that

$u w(w'/w) = uw(u'/uw)$ is a standard reduction of M. Then, u is leftmost-outermost with respect to u'.

Exercise 6.2.13 An *atomic label substitution* is a function $\mathtt{s}$ from atomic labels to $\mathbf{L}$. The previous function extends to generic labels in the natural way, i.e., to apply the atomic label substitution $\mathtt{s}$ to a label ℓ means to build the label $\ell[\mathtt{s}]$ obtained replacing each atomic label a of ℓ by the label $\mathtt{s}(a)$. The relabeling by $\mathtt{s}$ of a labeled term M is the term $M[\mathtt{s}]$ obtained from M by applying $\mathtt{s}$ to each label $\ell \in \mathbf{L}_0(M)$. The usual composition rule of functions induces in a natural way a notion of composition of atomic label substitutions, i.e., $\mathtt{s}\mathtt{s}'$ is the substitution such that $\ell[\mathtt{s}\mathtt{s}'] = (\ell[\mathtt{s}'])[\mathtt{s}]$, and analogously for relabeling of terms. Prove that:

(i) Any reduction $M^{\mathtt{I}} \stackrel{\rho}{\twoheadrightarrow} N$ induces an atomic label substitution $\mathtt{s}_\rho$ such that $N = N^{\mathtt{I}}[\mathtt{s}_\rho]$, and that $\mathsf{at}(\mathtt{s}_\rho(a)) \subset \mathbf{L}_0(M^{\mathtt{I}})$, for any $a \in \mathbf{L}_0(N^{\mathtt{I}})$.

(ii) $M[\mathtt{s}] \stackrel{\rho}{\twoheadrightarrow} N[\mathtt{s}]$ for any atomic label substitution $\mathtt{s}$ and any $M \stackrel{\rho}{\twoheadrightarrow} N$.

(iii) For any $\tau = \rho\sigma$, then $\mathtt{s}_\tau = \mathtt{s}_\rho \mathtt{s}_\sigma$.

Exercise 6.2.14 The atomic label substitution defined in the previous exercise can be extended to paths. Namely, if $M^{\mathtt{I}} \stackrel{\rho}{\twoheadrightarrow} N$, the *edge substitution* induced by the atomic label substitution $\mathtt{s}_\rho$ is a transformation $\phi[\mathtt{s}_\rho]$ of the paths of N such that $\mathsf{path}(\mathsf{degree}^{\rho}_{M}(u)) = \mathsf{path}(\mathsf{degree}^{\epsilon}_{N}(u))[\mathtt{s}_\rho]$. Give a formal definition of such a transformation of paths and prove that $\phi[\mathtt{s}_\rho]$ is a path of M, for any path ϕ of N, and that edge substitution accords to the composition law $\phi[\mathtt{s}_{\rho\sigma}] = \phi[\mathtt{s}_\rho \mathtt{s}_\sigma] = (\phi[\mathtt{s}_\sigma])[\mathtt{s}_\rho]$.(Warning: $\phi[\mathtt{s}_\rho]$ is not just the concatenation of the transformations of the edges of N. In fact, assuming to orient paths, an edge of N may appear in ϕ according its positive or to its negative orientation.)

6.2.3.3 Proof of the Contracted Label Lemma (Lemma 6.2.12)

Let us remind the hypotheses: $M \stackrel{w}{\to} N$; the two redexes $w\rho u$ and $w\sigma v$ are in canonical form; $\mathsf{degree}^{w\rho}_{M}(u) = \mathsf{degree}^{w\sigma}_{M}(v)$. We want to prove that such hypotheses imply $\mathsf{degree}^{\rho}_{N}(u) = \mathsf{degree}^{\sigma}_{N}(v)$.

Let $\phi = \mathsf{path}(\mathsf{degree}^{\rho}_{N}(u))$ and $\psi = \mathsf{path}(\mathsf{degree}^{\sigma}_{N}(v))$. By the definition of edge substitution given in Exercise 6.2.14, we have that $\mathsf{path}(\mathsf{degree}^{w\rho}_{M}(u)) = u[\mathtt{s}_w \mathtt{s}_\rho] = (u[\mathtt{s}_\rho])[\mathtt{s}_w] = \phi[\mathtt{s}_w]$, and analogously $\mathsf{path}(\mathsf{degree}^{w\sigma}_{M}(v)) = \psi[\mathtt{s}_w]$. By hypothesis, we have thus $\xi = \phi[\mathtt{s}_w] = \psi[\mathtt{s}_w]$. We want to prove that $\phi = \psi$.

Let $\mathfrak{a} = \mathsf{degree}^{\epsilon}_{M}(w)$. We start noticing that ξ cannot be internal to the argument of w, for by the First Redex Lemma we must have $\mathfrak{a} \in \mathsf{at}(\mathsf{degree}^{\rho}_{M}(\mathfrak{u}))$ (see also Exercise 6.2.15). To complete the proof we show that the following claim is true:

Let ϕ and ψ be two paths of N such that $\phi[\mathsf{s}_w] = \xi = \psi[\mathsf{s}_w]$. If ϕ and ψ are not internal to distinct instances of the argument of $\mathfrak{u}$, then $\phi = \psi$.

The proof of the claim is by induction on the occurrences of $\mathfrak{u}$ in ξ. In the base case, ξ does not contain $\mathfrak{u}$. Then, it is readily seen that we might have $\phi \neq \psi$ only if ϕ and ψ would be internal to distinct instances of the argument of $\mathfrak{u}$. But by hypothesis this is not the case. In the induction case, we see that chosen an arbitrary occurrence of w, there are two edges (atomic labels) $\mathfrak{c}$ and $\mathfrak{d}$ such that $\xi = \xi_1 \cdot \mathfrak{c}w\mathfrak{d} \cdot \xi_2$. Furthermore, we may assume without loss of generality that there is an atomic label (edge) $\mathfrak{b}$ of N^{I} such that $\mathsf{s}_w(\mathfrak{b}) = \mathfrak{c}\alpha\mathfrak{d}$, with $\alpha \in \{\underline{\mathfrak{a}}, \overline{\mathfrak{a}}\}$. In fact, if $\mathsf{s}_w(\mathfrak{b}) = \mathfrak{d}\alpha\mathfrak{c}$ we could just take the reverse of ξ (and then of ϕ and ψ). We distinguish two cases:

($\mathsf{s}_w(\mathfrak{b}) = \mathfrak{c}\overline{\mathfrak{a}}\mathfrak{d}$) In this case, $\mathfrak{c}$ is the context edge of $\mathfrak{u}$ and $\mathfrak{d}$ is the body edge. Thus, neither ξ_1 nor ξ_2 are internal to the argument of $\mathfrak{u}$: the first ends at the node above $\mathfrak{u}$, the second start in the body of the abstraction of $\mathfrak{u}$. So, by the induction hypothesis, there is a unique path ζ_1 and a unique path ζ_2 of N such that $\zeta_i[\mathsf{s}_w] = \xi_i$, with $i = 1, 2$. At the same time, the label (edge) $\mathfrak{b}$ is the unique one of N^{I} for which $\mathsf{s}_w(\mathfrak{b}) = \mathfrak{c}\overline{\mathfrak{a}}\mathfrak{d}$. We conclude then that $\phi = \zeta_1\mathfrak{b}\zeta_2 = \psi$.

($\mathsf{s}_w(\mathfrak{b}) = \mathfrak{c}\underline{\mathfrak{a}}\mathfrak{d}$) Here, $\mathfrak{c}$ is an edge connected to the binding port of the abstraction of $\mathfrak{u}$; while $\mathfrak{d}$ is the function edge of $\mathfrak{u}$, that is, the root edge of the argument of $\mathfrak{u}$. While it is still true that the label (edge) $\mathfrak{b}$ is the unique label of N^{I} for which $\mathsf{s}_w(\mathfrak{b}) = \mathfrak{c}\overline{\mathfrak{a}}\mathfrak{d}$ (note that in N, the labels of the roots of the instances of the argument of $\mathfrak{u}$ are all distinct) and that ξ_1 is not internal to the argument of $\mathfrak{u}$ (it ends in the body of $\mathfrak{u}$), the path ξ_2 might be internal to the argument of $\mathfrak{u}$. Anyway, there exists a unique path ζ_1 of N such that $(\zeta_1\mathfrak{b})[\mathsf{s}_w] = \xi_1 \cdot \mathfrak{c}\underline{\mathfrak{a}}\mathfrak{d}$. So, let ζ_2' and ζ_2'' be two paths such that $\xi = (\zeta_1\mathfrak{b}\zeta_2')[\mathsf{s}_w] = (\zeta_1\mathfrak{b}\zeta_2'')[\mathsf{s}_w]$. As the edge $\mathfrak{b}$ corresponds to a given instance of the argument of $\mathfrak{u}$, in the case that ξ_2 is internal to it, the paths ζ_2' and ζ_2'' are internal to the same instance of the argument of $\mathfrak{u}$. We can then apply the induction hypothesis concluding that $\zeta' = \zeta''$.

Exercise 6.2.15 Let $\sigma\nu$ be the canonical form of a redex with respect to the initial term M. Prove that if $\mathsf{path}(\mathsf{degree}^{\sigma}_{M}(\nu))$ is internal to the argument of a redex u of M, then the reduction $\sigma\nu$ is internal to the argument of u.

Exercise 6.2.16 Let $M \stackrel{\rho}{\twoheadrightarrow} N_1 \stackrel{u}{\rightarrow}$ and $M \stackrel{\sigma}{\twoheadrightarrow} N_2 \stackrel{\nu}{\rightarrow}$. If $\ell = \mathsf{degree}^{\sigma}_{M}(\nu)$, prove that $\mathsf{path}(\ell) \in \mathsf{degree}^{\rho}_{M}(u)$ or $\mathsf{path}(\ell)^{r} \in \mathsf{degree}^{\rho}_{M}(u)$ if and only if $\rho = \sigma'\nu'\tau'$, for some $\sigma'\nu'$ such that $\sigma'\nu' \simeq \sigma\nu$.

6.2.4 Well balanced paths and legal paths

The previous sections have given a good account of the correspondence between paths and labels. In particular, the proof of the correspondence between extraction relation and labeling has shown such a correspondence at work. Nevertheless, the only method so far available to recognize paths corresponding to virtual redexes is via the labeled calculus. That is, we can say that a path φ is a virtual redex if and only if there is a label ℓ generated along the reduction of M such that $\mathsf{path}(\ell) = \varphi$. The relevance of the set $\mathsf{path}(\mathbf{L}(M))$ of such paths is that it gives indeed another characterization of Lévy's families. In fact, because of the correspondence between families and degrees, each family correspond to a unique virtual redex (path) in M, vice versa, each virtual redex of M identify a unique family of redexes. Hence, once recognized the isomorphism between virtual redexes and families, the natural question is of finding a definition of $\mathsf{path}(\mathbf{L}(M))$ independent from the notion of labeled reduction and from the notion of extraction relation. In this way we would also get a new characterization of families independent from the dynamics of the calculus, or more precisely, we would get a characterization of the notions of history and dependency between redexes based on an implicit encoding of the λ-calculus reduction.

In this section we will show that such an implicit encoding of the dynamics is indeed possible and that it leads to the so-called *legal paths*.

Legal paths are defined in terms of another set of paths, the so-called *well balanced paths*, that at their turn are the language generated by a suitable set of context free composition rules. Having in mind the examples given to introduce the correspondence between paths and virtual redexes, the intuition should be clear: constructing the path of a virtual redex we work by "sessions", or equivalently, by nesting calls of a recursive algorithm; it is thus reasonable that the formalization

of such a naive procedure should lead to an inductive definition of the corresponding paths.

In more details, a session starts at the function port of an application moving along the corresponding edge. The purpose is to end at the context edge of an abstraction. So, the base case is when the initial edge is just a redex. In the other cases, the searching continues with a new session performing a recursive call of the algorithm. Then, in the case that the recursive subsession would end successfully, the initial session is resumed and the searching continues from the ending point of the result returned by the subsession.

Unfortunately, the context free approach corresponding to the previous algorithm gives raise to a set of paths wider than the ones corresponding to virtual redexes (by context free we mean that, applying the inductive step of the construction, we do not look inside the shape of the paths, but we just check if they start and end at the right ports). The problem has been already shown by one of the introductory examples: sometimes, moving from the binding port of an abstraction node, the search may continue only through some of the binding edges connected to the abstraction; in such cases, the choice of which binding edge we are allowed to cross depends on the shape of the path built so far.

Even if our concern is on paths connecting the function port of an application to the context port of an abstraction, for technical reasons we do not pose such a restriction in defining well balanced paths. In particular, well balanced paths still start at the function port of an application, but we distinguish them in three types, according to the port at which they end (i.e., a well balanced path can end at the context port of an applications, at the context port of an abstraction, or at the binding port of an abstraction). For the sake of a uniform presentation, we also assume that "the path φ ends at the context port of a variable node representing an occurrence of x" be indeed a synonym of "the path φ ends at the binding port the application representing λx". In other words, in this section we will freely swap between the representation of a syntax tree in which each occurrence of a variable is represented by a node, and the representation in which such nodes are omitted and the corresponding edges connected to the abstractions binding them. Finally, we will also say that an abstraction is a node of type λ, an application is a node of type @, and a variable is a node of type $\mathbf{v}$.

According to the previous notation, a well balanced path of type @-$\star$, with $\star \in \{@, \lambda, \mathbf{v}\}$, connects the function port of a node of type @ to the context port of a node of type $\star$. Namely, a well balanced path is:

- an @-@-*path*, when it ends at the context port of an application;
- an @-λ-*path*, when it ends at the context port of an abstraction;
- an @-v-*path*, when it ends at the context port of a variable (i.e., at the binding port of an application).

Definition 6.2.17 (wbp) The set of the *wbp*'s (well balanced paths) of a term M is inductively defined by the following rules (see also Figure 6.5):

Base case: The function edge of any application of M is a wbp.

@-*composition*: Let ψ be a wbp of type @-@, and φ be a wbp of type @-λ starting at the final application of ψ, then $\psi \cdot \varphi \cdot \mathfrak{u}$ is a wbp, where $\mathfrak{u}$ is the body edge of the final abstraction of φ.

λ-*composition*: If ψ is a wbp of type @-v, and φ is a wbp of type @-λ ending at the abstraction binding the variable at the end of ψ, then $\psi \cdot (\varphi)^{\mathrm{r}} \cdot \mathfrak{u}$ is a wbp, where $\mathfrak{u}$ is the argument edge of the initial application of φ.

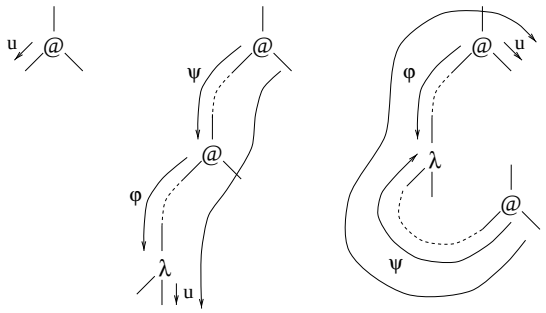


Fig. 6.5. Well balanced paths (*wbp*).

In the previous definition, every wbp of type @-λ corresponds to a session of the searching algorithm. Composition, that is, the execution of a subsession, explains thus how to resume the main session according to the way in which the subsession was entered. In particular, let us remark that composing paths we always add @-λ wbp's.

Another way to understand the definition of wbp's is comparing it with the decomposition of degrees in proper labels stated by Lemma 6.2.5. In fact, any subpath of type @-λ corresponds to an underlined or overlined label. Hence, according to Lemma 6.2.5, it must be surrounded by suitable atomic edges (see Figure 6.5).

Exercise 6.2.18 Verify that the interpretation of the wbp's of type @-λ as paths corresponding to degrees of redexes is compatible with the constraints in Lemma 6.2.5.

Example 6.2.19 Let us consider again the λ-term $(\Delta\,\Delta)$ with the labeling of Figure 6.6 (left). The edge $\mathtt{b}$ is a wbp of type @-λ. The edge $\mathtt{d}$ is a wbp of type @-ν. By λ-composition, $\mathtt{d}(\mathtt{b})^{\mathtt{r}}\mathtt{f} = \mathtt{dbf}$ is a wbp of type @-λ. It corresponds indeed to the redex created after one step along the unique derivation for $(\Delta\,\Delta)$. Let us go on computing paths. The edge $\mathtt{h}$ is a wbp of type @-ν and we have already built the wbp $\mathtt{dbf}$ leading from an application to the binder of the ending variable of $\mathtt{h}$. So, we can apply again λ-composition, obtaining the wbp $\mathtt{h}(\mathtt{dbf})^{\mathtt{r}}\mathtt{e} = \mathtt{hfbde}$ of type @-ν. By a further application of λ-composition, we finally get the wbp $\mathtt{hfbde}(\mathtt{b})^{\mathtt{r}}\mathtt{f} = \mathtt{hfbdebf}$ of type @-λ, that is the path associated to the unique redex created after two β-reductions.
Up to now, the correspondence between virtual redexes and wbp's is complete: any wbp we have found so far is a virtual redex; vice versa, the virtual redexes corresponding to the first yielded by first two steps of the reduction of $(\Delta\,\Delta)$ are wbp's. Unfortunately, such a correspondence is lost at the next step. First of all, let us note that the result of the previous computation are two @-λ-paths leading to the same application marked with $\mathtt{f}$; namely, $\varphi = \mathtt{dbf}$ and $\psi = \mathtt{hfbdebf}$. By λ-composition, $\mathtt{h}(\psi)^{\mathtt{r}}\mathtt{k}$ is then an @-ν-path whose final variable is bound by the λ marked by $\mathtt{f}$. According to the paths built so far, the construction can then proceed in two ways: (*i*) following φ; (*ii*) repeating ψ an arbitrary number of times. In the first case, we would end up with $\mathtt{h}(\psi)^{\mathtt{r}}\mathtt{k}(\varphi)^{\mathtt{r}}\mathtt{ebf}$, that corresponds to the redex created after three β-reductions. It the second case, we would rather construct paths with the shape

$$\mathtt{h}(\psi)^{\mathtt{r}}\mathtt{k}(\psi)^{\mathtt{r}}\mathtt{k}\cdots(\psi)^{\mathtt{r}}\mathtt{k}(\varphi)^{\mathtt{r}}\mathtt{ebf}$$

and none of them corresponds to a redex.

Example 6.2.20 A simpler example of the same phenomenon is provided by the term of Example 6.2.1 (redrawn on the right hand side of Figure 6.6). There are two wbp's, $\varphi = \mathtt{fbl}$ and $\psi = \mathtt{hbl}$, leading to the same λ. The two paths $\mathtt{d}\varphi\mathtt{m}(\varphi)^{\mathtt{r}}\mathtt{g}$ and $\mathtt{d}\varphi\mathtt{m}(\psi)^{\mathtt{r}}\mathtt{k}$ are both well balanced, but only the first one is "legal".

As we already remarked, the only non-determinism in the algorithm

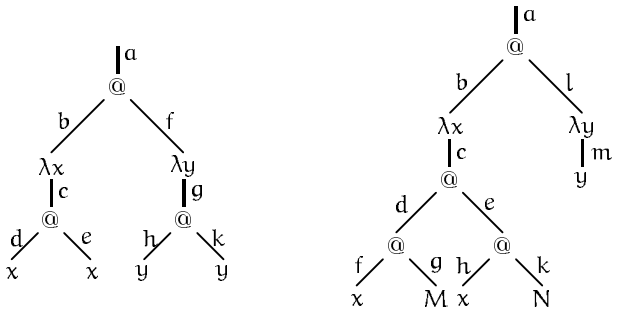


Fig. 6.6. Left: $(\Delta\,\Delta)$. Right: $(\lambda x.((x\,M)\,(x\,N))\,\lambda y.y)$.

searching virtual redexes (in the definition of wbp) is at the level of bound variables. Furthermore, the latter examples clearly points out that not always such nondeterminism is unconstrained. In fact, according to the λ-composition rule, any binding edge ν occurring in a wbp is eventually followed by a (reversed) wbp φ of type @-λ, and by an access to the argument N of the initial application of φ. The shape of φ might be very complex; in particular, it might describe a cycle internal to N (Figure 6.7). Let us assume that this is the case. Intuitively, the path φ should travel inside the instance of N replacing the variable occurrence corresponding to ν. So, after the cycle internal to N, a legal path "cannot jump" inside another instance of N, it is rather forced to follow back the same path traversed to access N before the cycle (see Example 6.2.20).

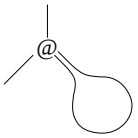


Fig. 6.7. Naive definition of cycle

The main problem in formalizing the previous intuition is to capture the "right" notion of cycle. The first idea would be to define cycles according to Figure 6.7, that is, without any other traversal of their initial application node. Unfortunately, this naive approach fails, for the path might traverse several times the function port of the initial @ node (i.e., according to Figure 6.8) before looping back to its argument

port—furthermore, the crossings in Figure 6.8 might neither correspond to an inner occurrence of a cycle of the same type.

Fig. 6.8.

In other words:

(i) Cycles may contain occurrences of other cycles, even of cycles looping to the same port;
(ii) It is generally incorrect to find the cycles in a path pairing a point in which the path exits from the argument of an application to the last point in which the path entered into it (i.e., according to a last-in-first-out strategy).

According to the latter remarks, the base case of the inductive definition of cycles is immediate: we can start from the paths *physically* internal to the argument of an application.

Definition 6.2.21 (elementary cycle) Let φ be a wbp. A subpath ψ of φ is an *elementary* @*-cycle* of ψ (over an @-node @) when:
(i) ψ starts and ends with the argument edge of the @-node @;
(ii) ψ is internal to the argument N of the application corresponding to the @-node @ (i.e., ψ does not traverse any variable that occurs free in N).

The definition of the inductive step requires instead more regards.. The problem is that traveling along the cycle we could exit from some free variable of N (N, but eventually bound in the initial term), make a cycle somewhere else, and come back again inside N. According to this, we need to introduce a notion of $\mathbf{v}$*-cycle* (i.e., a cycle looping over a variable) describing such a situation. Furthermore, for the sake of clarity, we will write $\varphi\star\psi$ to denote that the paths φ and ψ respectively ends and start at a node of type $\star$.

Definition 6.2.22 (cycle) Let ζ be a wbp. The set of the @-cycles of ζ (over an @-node @) and of the $\mathbf{v}$-cycles of ζ (over the occurrence ν of a variable) is inductively defined as follows:

Elementary: Every elementary @-cycle of ζ (over an @-node @) is an @-cycle.

$\mathtt{v}$-*cycle*: Every cyclic subpath of ζ of the form $\nu\,\lambda\,(\varphi)^{\mathrm{r}}\,@\,\psi\,@\,\varphi\,\lambda\,\nu$, where φ is a wbp, ψ is an @-cycle and ν is a binding edge, is a $\mathtt{v}$-cycle (over ν).

@-*cycle*: Every subpath ψ of ζ that starts and ends with the argument edge of a given @-node @, and that is composed of subpaths internal to the argument N of @ and $\mathtt{v}$-cycles over free variables of N is an @-cycle (over the @-node @).

In the definition of @-cycles, we have eventually inserted a consistency condition requiring that a path looping to the argument of an application must be accessed and exited though the same wbp. The next step is to extend such a constraint to all the cycles occurring in a wbp. Namely, that any @-cycle is surrounded by a consistent pair of wbp's. However, before to do this, let us verify that the decomposition of a wbp into smaller wbp's is unique.

Proposition 6.2.23 *Let φ be a wbp.*

(i) *For every λ-node in φ there exists a unique wbp ζ_2 (of type @-λ) such that φ can be decomposed as $\zeta_1\,@\,\zeta_2\,\lambda\,\zeta_3$ or $\zeta_1\,\lambda\,(\zeta_2)^{\mathrm{r}}\,@\,\zeta_3$. Furthermore, the given λ-node is paired with the initial @-node of φ if and only if it is the last node of φ.*

(ii) *For every @-node in φ but the initial one, there exists a unique wbp ζ_2 (of type @-λ) such that φ can be decomposed as $\zeta_1\,@\,\zeta_2\,\lambda\,\zeta_3$ or $\zeta_1\,\lambda(\zeta_2)^{\mathrm{r}}\,@\,\zeta_3$.*

Proof By easy induction on the definition of wbp. □

Remark 6.2.24 In both the cases of the previous proposition, the shape of the decomposition is fixed by the way in which φ crosses the corresponding node, e.g., we can have $\zeta_1\,@\,\zeta_2\,\lambda\,\zeta_3$ only if φ crosses the λ-node entering through its context (i.e., its principal) port, and crosses the @-node exiting through its argument port. Furthermore, when φ is of type @-λ, the initial application is always paired with the final abstraction of the node.

Exercise 6.2.25 Let ψ_1 and ψ_2 be two wbp's contained in the wbp φ. Prove that ψ_1 and ψ_2 overlap only if one is a subpath of the other, i.e., either ψ_1 and ψ_2 are disjoint in φ, or ψ_2 is a subpath of ψ_1 (assuming w.l.o.g. that ψ_1 is not shorter than ψ_2).

A consequence of the previous proposition is that any @-cycle is surrounded by two wbp's of type @-λ.

Corollary 6.2.26 *Let ψ be an @-cycle of φ over an @-node @. The wbp φ can be uniquely decomposed as*

$$\zeta_1 \lambda (\zeta_2)^r @ \psi @ \zeta_3 \lambda \zeta_4$$

where ζ_2 and ζ_3 are wbp's of type @-λ.

Proof Let us apply Proposition 6.2.23 to the occurrence of @ at the beginning of ψ and to the occurrence at the end of ψ. According to Remark 6.2.24, the unique decomposition we can get is the one above. In fact, the occurrence of ψ in φ is preceded and followed by the argument edge of @. □

In the situation of Corollary 6.2.26, the wbp ζ_2 and the wbp ζ_3 are respectively the *call* and *return* paths of the @-cycle ψ. The last label of ζ_1 and the first label of ζ_4 are instead the *discriminants* of the call and return paths, respectively (note that discriminants are binding edges connected to final λ-node of the call and return paths).

We are now ready to state the *legality* condition for wbp's.

Definition 6.2.27 (legal path) A wbp is a *legal path* if and only if the call and return paths of *any* @-cycle are one the reverse of the other and their discriminants are equal.

In particular, we will say that an @-cycle is *legal*, when its call and return parts accord with the legality constraint in the previous definition. Let us also note that an @-cycle is legal only when it can be used to build a v-cycle. Hence, in a legal path any @-cycle is contained inside some v-cycle.

Example 6.2.28 Let us consider again the wbp's dfblmlbfg and dfblmlbhk of Example 6.2.20. Both contain an (elementary) @-cycle lml over the @-node labeled with a. In both cases the call and return path of lml is b. Nevertheless, only the first path is legal, for in the second case the discriminants differ (they are f and h).

To conclude our analysis of the definition of legal paths, let us provide a detailed counter-example to the naive definition of cycle (i.e., to the fact that all the @-cycles are elementary).

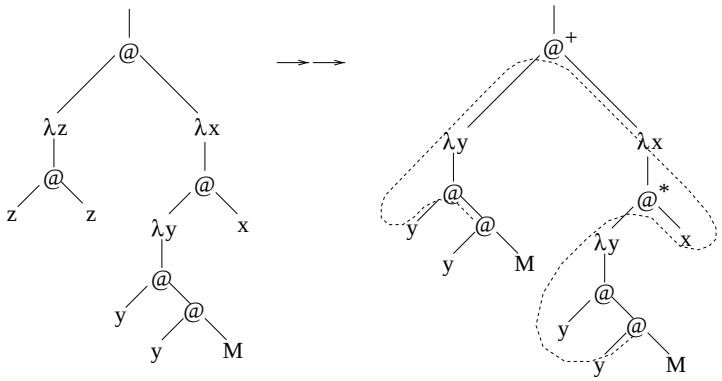


Fig. 6.9. $(\Delta\ N) \twoheadrightarrow (\lambda y.(y\ (y\ M))\ N)$, with $N = \lambda x.(\lambda y.(y\ (y\ M))\ x)$.

Example 6.2.29 Let us take (see Figure 6.9) the terms $P = (\Delta\ N)$ and $Q = (\lambda y.(y\ (y\ M))\ N)$, where $N = \lambda x.(\lambda y.(y\ (y\ M))\ x)$. It is readily seen that $P \to (N\ N) \to Q$. The applications respectively marked with $+$ and $*$ in Figure 6.9 are both residual of a same application in P. Moreover, the dotted path in Figure 6.9 is obviously a legal path of type @-@. This path enters the argument of $@^*$ and immediately exit from the argument of $@^+$. Since these two applications are residual of a same application in P, the ancestor of the above portion of path would be a cycle in the naive sense, that is instead obviously wrong. Explicitly, the ancestor in P of the dotted path in Q is described in Figure 6.10: we have a "naive" cycle inside the argument of $@^\bullet$, but this is not an @-cycle in the sense of Definition 6.2.22 (prove it as an exercise).

6.2.5 Legal paths and redex families

We shall now prove than there is a bijective correspondence between legal paths and paths yielded by degrees. More formally, we shall prove that $\mathsf{path}(\mathbf{L}(T))$ is the set of the legal paths of T.

Some preliminary definitions and results are required.

Definition 6.2.30 Let φ be a wbp in a term T. The label of φ is defined inductively as follows:

(*base case*) When $\varphi = u$, the label of φ is equal to the label of u in T^{I}.

(@-*composition*) When $\varphi = \varphi_1 @ \varphi_2\, \lambda u$, the label of φ is $\ell_1\, \overline{\ell_2}\, a$, where

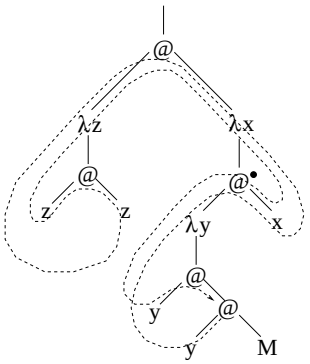


Fig. 6.10. An example of "naive" loop that is not a cycle.

ℓ_1 is the label of φ_1, ℓ_2 is the label of φ_2, and $\mathfrak{a}$ is the label in T^{I} of $\mathfrak{u}$.

(λ-*composition*) When $\varphi = \varphi_1 \lambda \varphi_2 @ \mathfrak{u}$, the label of φ is $\ell_1 (\underline{\ell_2})^{\mathrm{r}} \mathfrak{a}$, where ℓ_1 is the label of φ_1, ℓ_2 is the label of φ_2, and $\mathfrak{a}$ is the label in T^{I} of $\mathfrak{u}$.

According to the above definition, it is immediate to verify that $\varphi = \mathsf{path}(\ell)$, when ℓ is the label of a wbp φ.

Let $\mathsf{T} \xrightarrow{\mathfrak{u}} \mathsf{T}'$. Every (arbitrary) path in T' has an *ancestor* in T. Its definition is pretty intuitive, so we shall not be pedantic. In particular, we shall just define the notion of ancestor for single edges in T'. This extends to paths by composition, in the obvious way.

Let $\mathsf{T} = \mathsf{C}[(\lambda x.\, \mathsf{M}\ \mathsf{N})]$ and $\mathsf{T}' = \mathsf{C}[\mathsf{M}[{}^{\mathsf{N}}/_{x}]]$, where $\mathsf{C}[\,.\,]$ is some context. Let us note first that some edges in T' are "residuals" of edges in T. This is the case for every edge ν' belonging to the context, or internal to M or to some instance of N. If ν' is a residual of ν in the previous sense, then ν is the ancestor of ν'. The problem is when ν' is a new connection created by firing the redex $\mathfrak{u}$. Let @ and λ be the two nodes connected by $\mathfrak{u}$ in T. Three cases are possible (see also Figure 5.8):

- ν' is a connection between the context $\mathsf{C}[\,.\,]$ and the body M. The ancestor of ν' is $\mathfrak{a} \cdot \mathfrak{u} \cdot \mathfrak{b}$, where $\mathfrak{a}$ is the connection from the context to @ (i.e., the context edge of @), and $\mathfrak{b}$ is the connection from λ to M (i.e., the body edge of λ).
- ν' is a connection between M and the i-th instance of N. The ancestor of ν' is $\mathfrak{b} \cdot (\mathfrak{u})^{\mathrm{r}} \cdot \mathfrak{c}$, where $\mathfrak{b}$ is the edge leading from M to the i-th instance of the variable bound by λ (i.e., an edge corresponding to an

occurrence of the variable x), and c is the edge leading from @ to N (i.e., the argument edge of @).

- ν' is a connection between the context and N. This is only possible when $M = x$, and it is an obvious combination of the previous cases. In particular, the ancestor of ν' is $a \cdot u \cdot b \cdot (u)^r \cdot c$, with a, b and c as above.

We leave to the reader the care to generalize the definition of ancestor from edges to arbitrary paths—the only thing to prove is that composing the ancestors of the edges in a path the result is a connected sequence. The definition of ancestor is then extended to derivations in the usual way (transitive and reflexive closure of the base case). Moreover, the ancestor relation preserves path balancing.

Proposition 6.2.31 *Every ancestor of a wbp is a wbp.*

Proof By induction on the length of the derivation leading to the term in which the wbp occurs. □

Lemma 6.2.32 *Let* $T \xrightarrow{u} T'$. *If* φ *is the ancestor of a legal path* φ' *in* T', *then:*

(i) *Every* @*-cycle in* φ *relative to an application* @ *not fired by* u *is ancestor of an* @*-cycle in* φ' *over any residual of* @.

(ii) *Every* v*-cycle in* φ *relative to an instance* x *of a variable not bound by the* λ *in* u *is ancestor of a* v*-cycle in* φ' *over any residual of* x.

Proof By induction on the definition of cycle. Let ψ be the cycle under analysis. We have the following cases:

(*elementary*) Trivial, for the execution of u does not change the fact that ψ is internal to the argument of its initial @-node.

(v*-cycle*) Let $\psi = \nu\,\lambda\,(\varphi_1)^r\,@\,\xi\,@\,\varphi_1\,\lambda\,\nu$. Since ν is not bound by the λ-node in u, the @-cycle ξ cannot be relative to the application fired by u (indeed, u is the unique wbp in T relative to this application). So, applying the induction hypothesis, we see that there exists a cycle ξ' inside φ' whose ancestor is ξ. By hypothesis, φ' is legal. Hence, by Corollary 6.2.26, it contains a unique subpath such that $\nu'\,\lambda\,(\varphi'_1)^r\,@\,\xi'\,@\,\varphi'_1\,\lambda\,\nu'$. Let $\widetilde{\nu}\,\lambda\,(\widetilde{\varphi}_1)^r\,@\,\xi\,@\,\widetilde{\varphi}_1\,\lambda\,\widetilde{\nu}$ be the ancestor of the previous subpath

in T (i.e., $\widetilde{\varphi}_1$ is the ancestor of φ_1). By Proposition 6.2.31, $\widetilde{\varphi}_1$ is a wbp. Then, by Corollary 6.2.26, $\varphi_1 = \widetilde{\varphi}_1$ and $\nu = \widetilde{\nu}$.

(*@-cycle*) Let $@_*$ be the @-node over which ψ loops. By definition, ψ composes of an alternated sequence of paths such that

$$\psi = \xi_1 \cdot \nu_1 \, \lambda \, \psi_1 \, \lambda \, \nu_1 \cdot \xi_2 \cdots \xi_k \cdot \nu_k \, \lambda \, \psi_k \, \lambda \, \nu_k \cdot \xi_{k+1}$$

where, any ξ_i is internal to the argument M of $@_*$, while any $\nu_i \, \lambda \, \psi_i \, \lambda \, \nu_i$ is a $\mathfrak{v}$-cycle over a variable of M.
Firstly, let us assume that none of the edges ν_i is connected to the λ-node in the redex $\mathfrak{u}$. In this case, we have that:

$$\psi' = \xi'_1 \cdot \nu'_1 \, \lambda \, \psi'_1 \, \lambda \, \nu'_1 \cdot \xi'_2 \cdots \xi'_k \cdot \nu'_k \, \lambda \, \psi'_k \, \lambda \, \nu'_k \cdot \xi'_{k+1}$$

where, for any ξ'_i, the path ξ_i is its ancestor, and for any $\nu'_i \, \lambda \, \psi'_i \, \lambda \, \nu'_i$, the $\mathfrak{v}$-cycle $\nu_i \, \lambda \, \psi_i \, \lambda \, \nu_i$ is its ancestor. We claim that all the ξ'_i are internal to the argument M' of the @-node $@'_*$ over which ψ' loops. The only relevant case is when some ξ_i contains a node in $\mathfrak{u}$. When this happens, it is immediate to check that ξ_i crosses the redex $\mathfrak{u}$, that is, $\mathfrak{u}$ is internal to M. The claim is then an immediate consequence of the previous remark. Thus, to conclude the proof for the case no loop over the variable of $\mathfrak{u}$, let us note that any $\nu'_i \, \lambda \, \psi'_i \, \lambda \, \nu'_i$ is a $\mathfrak{v}$-cycle over a variable of M', by induction hypothesis. That is, ψ' is an @-cycle.
Now, let us assume that ν_i is a binding edge connected to the λ-node in $\mathfrak{u}$. The redex $\mathfrak{u}$ is the first and last edge of ψ_i. Moreover, $\mathfrak{u}$ is external to M, otherwise we could take a longer path ξ_i contained in M. By the definition of $\mathfrak{v}$-cycle and by Corollary 6.2.26, we see that $\psi_i = \nu_i \, \lambda \, \mathfrak{u} \, @ \, \zeta @ \, \mathfrak{u} \, \lambda \nu_i$, for some @-cycle ζ (again, remind that $\mathfrak{u}$ is an @-λ wbp). The @-cycle ζ can be decomposed at its turn in paths internal to the argument N of $\mathfrak{u}$ and $\mathfrak{v}$-cycles over variables of N. Moreover, let us note that: (*i*) none of such $\mathfrak{v}$-loops can be over the variable bound by the λ-node of $\mathfrak{u}$; (*ii*) since M is internal to the body of $\mathfrak{u}$ (by the syntax of λ-terms), @ has a unique residual $@_*$; (*iii*) the residual in ψ'_i of the paths internal to N are paths internal to the instance of N replacing the occurrence of the variable corresponding to ν_i. As a consequence, we see that the residual of ψ_i is a path composed of subpaths internal to the argument M' of $@_*$, and of residuals of $\mathfrak{v}$-cycles over variables of N that,

by induction hypothesis and by (*iii*), are $\mathbf{v}$-cycles over variables in M'. This suffices to conclude.

□

Lemma 6.2.33 *Let* $T \xrightarrow{u} T'$. *If* φ *is the ancestor of a* legal *path* φ' *in* T', *then every* @*-cycle* ψ *in* φ *relative to an application not fired by* u *is legal.*

Proof By Lemma 6.2.32, ψ is ancestor of an @-cycle ψ' in φ'. Since φ' is legal, φ' contains a $\mathbf{v}$-cycle $\nu' \lambda (\varphi'_1)^r @ \psi' @ \varphi'_1 \lambda \nu'$. So, the structure of φ around the @-cycle ψ has the shape $\nu \lambda (\varphi_1)^r @ \psi @ \varphi_1 \lambda \nu$, where φ_1 is the ancestor of φ'_1, and ν' is the ancestor of ν. □

Lemma 6.2.34 *Let* $T \xrightarrow{u} T'$. *If* φ *is the ancestor of a* legal *path* φ' *in* T', *then every* @*-cycle* ψ *in* φ *relative to the application fired by* u *is legal.*

Proof The call and return paths of ψ are equal to u, that is, they are trivially equal. Thus, let $\nu_1 \lambda u @ \psi @ u \lambda \nu_2$. We must prove that the discriminants ν_1 and ν_2 are equal. By definition, ψ composes of subpaths internal to the argument N of the redex u and ν-cycles over free variables of N. By Lemma 6.2.32, each of such ν-cycles is the ancestor of a ν-cycle in φ'. Furthermore, such cycles in φ' are relative to free variables of some instance N_i of N in T (see the proof of Lemma 6.2.32). Let us note that: (*i*) all the instances of N are disjoint in T'; (*ii*) the context edge of each N_i corresponds to a unique discriminant of the variable of u. Hence, by induction on the number of $\mathbf{v}$-cycles in ψ, we conclude that all the residuals of the paths internal to N in ψ are in the same instance N_i, and that all the residuals of the $\mathbf{v}$-cycles in ψ are over free variables in that instance N_i. As a consequence, the path $\nu_1 \lambda u @ \psi @ u \lambda \nu_2$ is the ancestor of a path ξ starting and ending at the context edge of N_i. Thus, we conclude that $\nu_1 = \nu_2$. □

The previous lemmas ensure that legal paths cannot be created along reduction.

Proposition 6.2.35 *Let* $T \xrightarrow{u} T'$. *The ancestor of a legal path* φ' *in* T' *is a legal path* φ *in* T.

Proof By Proposition 6.2.31 we know that φ is a wbp. By the previous lemmas we see that it is indeed legal. □

A relevant point for the correspondence between legal paths and degrees is that a legal path and any of its ancestors yield the same label.

Proposition 6.2.36 *Labeling of legal paths is unchanged in ancestors.*

Proof By induction on the definition of ancestor and by analysis of the shape of the edges created at any reduction given in page 176. □

We are now ready to prove that the set of the legal paths in a term T contains the set $\mathsf{path}(\mathbf{L}(\mathsf{T}))$ of the paths yielded by the degrees obtainable reducing T. That is the equivalence between degrees and legal paths in the direction from degrees to legal paths.

Theorem 6.2.37 (from degrees to legal paths) *Every path yielded by the degree of a redex is a legal path.*

Proof Any redex u in a term T' is a legal path. By Proposition 6.2.35 and Proposition 6.2.36, if $\mathsf{T} \overset{\rho}{\twoheadrightarrow} \mathsf{T}'$, then the ancestor of u in T is a legal path φ whose label ℓ is equal to $\mathsf{degree}^{\rho}_{\mathsf{T}}(\mathsf{u})$. Finally, as $\varphi = \mathsf{path}(\ell)$, we conclude. □

The strategy to show the converse of Theorem 6.2.37 bases on showing that any legal path can be uniquely "contracted" into a legal path by firing the leftmost-outermost redex. By the previous results, we know that such a contraction preserves legality; furthermore, since it also decreases the length of the path, we can conclude that iterating the process we eventually get a legal path corresponding to the base case in the definition of wbp; that is a redex, when the path is of type @-λ.

Some preliminary results are required.

Lemma 6.2.38 *Let* $\mathsf{T} \overset{\mathsf{u}}{\to} \mathsf{T}'$, *with* $\mathsf{u} = (\lambda \mathsf{x}.\, \mathsf{M}\ \mathsf{N})$. *Let* φ *be a legal path internal to* N. *Each instance* N_i *of* N *in* T' *contains a legal path* φ_i *whose ancestor is* φ.

Proof By inspection of the β-rule. □

Lemma 6.2.39 *Let* $\mathsf{T} \overset{\mathsf{u}}{\to} \mathsf{T}'$, *with* $\mathsf{u} = (\lambda \mathsf{x}.\, \mathsf{M}\ \mathsf{N})$. *Let* φ *be any legal path in* T *such that* u *is the leftmost-outermost redex that* φ *traverses in* T. *Then, when* φ *does not start or end at any port of a node in* u, *there is a* unique *wbp* φ' *in* T' *whose ancestor is* φ.

Proof By induction on the structure of φ. The base case holds trivially. In fact, since φ must contain the redex u, the only possibility when $|\varphi| = 1$ is $\varphi = u$. That is, φ violates the "when" clause of the thesis.
Let us now proceed with the induction case ($|\varphi| > 1$). The only problem is to prove that suitably choosing the residuals of the edges in φ we get a connected sequence in T', that is, a path φ'.
Let $\varphi = \varphi_1 @ \varphi_2 \lambda\nu$, i.e., φ is the result of an @-composition. We distinguish three cases according to the position of the @-node @ between φ_1 and φ_2 with respect to N:

(i) @ *is external to* N: The path φ_2 cannot end at the λ-node of u, for otherwise we would have $\varphi_2 = u$. Furthermore, if the "when" clause" of the thesis holds, then φ_1 does not start at the @-node of u. According to this, we see that the paths φ_1 and φ_2 determine a unique pair of wbp's φ_1' and φ_2' such that φ_i is ancestor of φ_i', for $i = 1,2$. In fact, for both φ_1 and φ_2 we can distinguish two subcases: (*i*) φ_i contains u; (*ii*) φ_i does not contain u, i.e., it is external to N. In case (*i*), our preliminary considerations ensure that we can apply the induction hypothesis. In case (*ii*), by the definition of β-rule, φ_i is ancestor of a unique path φ_i' in T'; furthermore, since φ_i' and φ_i are isomorphic, φ_i' is a wbp. To conclude, let us note that @ and ν have unique residuals in T': the node $@'$ at the end (beginning) of φ_1' (φ_2'), and the edge ν', respectively. Hence, $\varphi' = \varphi_1' @ \varphi_2' \lambda\nu'$ is a wbp.

(ii) @ *is internal to* N: As in the previous case, we can exclude that φ_2 ends at the λ-node of u, and assume that φ_1 does not start at the @-node of u. As in the previous case we have to distinguish whether φ_i traverses u or not. The only difference is that here, in subcase (*ii*), the path φ_i is completely internal to N. Let us however note that case (*ii*) cannot simultaneously hold for both φ_1 and φ_2. Hence, the induction hypothesis applies to at least one of the paths. For instance, let us assume that φ_2 is internal to N. The path φ_1' ends at a residual $@'$ of @ in T' internal to a given instance N' of N. By Lemma 6.2.38, in any instance N_i of N in T', there is a wbp φ_2^i such that φ_2 is ancestor of φ_2^i. Nevertheless, the only φ_2^i that yields a connected sequence is the instance φ_2' internal to N'. By construction, we have then that $\varphi' = \varphi_1' @ \varphi_2' \lambda\nu'$ is a wbp.
The case φ_1 internal to N is similar. Then, we have left to prove the case in which the induction hypothesis applies to both φ_1 and

φ_2. Namely, we have to prove that, when both φ_1 and φ_2 contain u, the sequence $\varphi_1' @ \varphi_2' @ u'$ obtained composing the (unique) paths associated to φ_1 and φ_2 is indeed a path. This fact is not trivially immediate, for φ_1' might end at a residual $@^*$ of $@$ in an instance N^* of N, and φ_2' might start at the residual $@^+$ of $@$ in an instance N^+ of N. Nevertheless, take the last instance of u in φ_1 and the first instance in φ_2, we have $\varphi_1 = \psi_1 \cdot u \cdot \xi_1$ and $\varphi_2 = \xi_2 \cdot u \cdot \psi_2$. Furthermore, since both ξ_1 and ξ_2 must be internal to N, ξ_1 starts at the argument port of the @-node of u, while ξ_2 ends at this port. As a consequence, $\xi = \xi_1 \cdot \xi_2$ is an elementary case of @-cycle. The call and return paths of ξ are u and, by hypothesis, their discriminants coincide and are are equal to an occurrence of x. Hence, since any occurrence of x determines a unique instance of N in T', we conclude that $N^* = N^+$, that is, $@^* = @^+$.

(iii) $@$ *is the* @*-node of* u: Thas is, $\varphi_2 = u$. First of all, let us note that the last edge of φ_1 is definitely the context edge w of the redex u. Then, let us proceed by induction on the shape of φ_1:

- ($\varphi_1 = w$) The only case in which the "when" clause of the thesis is violated is when v is connected to the binding port of the λ-node in u (that is $M = x$). Hence, let us assume that this is not the case. There is a unique edge v' of T' whose ancestor is $\varphi = wuv$. Since such edge starts at the unique residual in T' of the @-node at the beginning of φ, we conclude that v' is a wbp.
- ($\varphi_1 = \psi_1 @ \psi_2 \lambda w$) The path ψ_2 ends at a λ-node immediately above the @-node of u. Then the @-node between ψ_1 and ψ_2 cannot be the application of u. Using the same technique of the case "@ external to N", we can then conclude that $\psi_1 @ \psi_2$ is ancestor of a unique path $\psi_1' @ \psi_2'$ composed of two wbp's ψ_1' and ψ_2'. Hence, if v' is the residual of wuv, we see that $\varphi_1' @ \varphi_2' \lambda v'$ is the unique wbp in T' whose ancestor is φ.
- ($\varphi_1 = \psi_1 \lambda (\psi_2)^r @ w$) As in the previous case we can exclude that ψ_1 and ψ_2 ends at the λ-node of u. Then, using the same reasoning, we conclude.

The case of λ-composition (i.e., $\varphi = \varphi_1 \lambda (\varphi_2)^r @ v$) is totally similar to the case of @-composition. We leave it to the reader as an easy exercise. □

Lemma 6.2.40 *Let* φ, T, $u = (\lambda x.\, M\ N)$, T', *and* φ' *be as in the previous lemma. For any cycle* ψ' *contained in* φ *there is a cycle* ψ *in* φ *such that* ψ *is the ancestor of* ψ'.

Proof Similar to the proof of Lemma 6.2.32. The difficult point is in the case of @-cycle, where the construction used in Lemma 6.2.32 must be reverted. Let us proceed by induction on φ'.

(*elementary*) Trivial.

($\mathfrak{v}$-*cycle*) Let $\psi' = \nu' \lambda\, (\varphi_1')^r \mathbin{@} \xi' \mathbin{@} \varphi_1' \lambda \nu'$. The @-cycle ξ' is not relative to the application fired by u. Applying the induction hypothesis, the ancestor of ξ' in φ is an @-cycle ξ. By hypothesis, φ is legal. Hence, by Corollary 6.2.26, it contains a subpath $\psi = \nu \lambda\, (\varphi_1)^r \mathbin{@} \xi \mathbin{@} \varphi_1 \lambda \nu$, where φ_1 is a legal path. By Lemma 6.2.39 and Corollary 6.2.26, φ_1 is the unique ancestor of φ_1'. Thus, ψ is the ancestor of ψ'.

(@-*cycle*) Let @′ be the @-node over which ψ' loops. By definition, ψ composes of an alternated sequence of paths such that

$$\psi' = \xi_1' \cdot \nu_1' \lambda \psi_1' \lambda \nu_1' \cdot \xi_2' \cdots \xi_k' \cdot \nu_k' \lambda \psi_k' \lambda \nu_k' \cdot \xi_{k+1}'$$

where, any ξ_i' is internal to the argument P' of $@_*'$, while any $\nu_i' \lambda \psi_i' \lambda \nu_i'$ is a $\mathfrak{v}$-cycle over a variable of P'. Correspondingly, in φ there is a path

$$\psi = \xi_1 \cdot \nu_1 \lambda \psi_1 \lambda \nu_1 \cdot \xi_2 \cdots \xi_k \cdot \nu_k \lambda \psi_k \lambda \nu_k \cdot \xi_{k+1}$$

where, any ξ_i is the ancestor of the corresponding ξ_i', and any $\nu_i \lambda \psi_i \lambda \nu_i$ is the ancestor of the corresponding $\mathfrak{v}$-cycle $\nu_i' \lambda \psi_i' \lambda \nu_i'$. In particular, by Lemma 6.2.39 any $\nu_i \lambda \psi_i \lambda \nu_i$ is a $\mathfrak{v}$-cycle. The easy case is when none of the ξ_i crosses u. If this is the case, it is indeed immediate to see that all the ξ_i are internal to the argument P of the @-node @ at the beginning of ψ, and that all the $\mathfrak{v}$-cycles are over variables free in P. Thus, let us analyze the other situation.

The edge u may occur in some ξ_i only when the variable x of u occurs in M. According to this we have to distinguish two subcases: (*i*) u occurs in P; (*ii*) the node @ and then P are in M. In case (*i*), it is immediate that any ξ_i is internal to P and, as a consequence, thatψ is an @-cycle. In case (*ii*) instead, let us take the first i such that ξ_i contains u. The only possibility is that $\xi_i = \chi_i \cdot w \lambda u \mathbin{@} v \zeta_i$, where w is an occurrence of x and

ν is the context edge of N. According to this, after χ_i, the path ψ exits from P and enters into N, but, since the last node of ψ is in P, ψ must eventually reenter into N. Let us see how this can happen; in particular, the following steps will show how to build an @-cycle starting from the given occurrence of u. In fact:

(a) no variable free in N is bound by a λ-node in P;

(b) since ζ_i is ancestor of a path ζ'_i contained in P, ζ_i cannot exit N through any variable free in N;

(c) when ζ_i is completely internal to N, ζ'_i is internal to the instance N_w of N corresponding to the discriminant w;

(d) when ζ_i is completely internal to N, the successive $\mathbf{v}$-cycle $\nu_i \lambda \psi_i \lambda \nu_i$ is ancestor of a $\mathbf{v}$-cycle over a variable occurring free in N_w, then ξ'_{i+1} starts at a λ-node internal to N_w;

(e) as for ζ_i, neither ξ_{i+1} can exit from N through a variable occurring free in N;

(f) iterating the previous reasonings we finally reach some ξ_j that exits from N through the redex u, that is, $\xi_j = \zeta_j \cdot \nu \lambda u @ \widetilde{w} \cdot \chi_j$;

(g) the result is a subpath $\xi_i \cdots \xi_j = \chi_i \cdot w \lambda u @ \nu \cdot \zeta \cdot \nu \lambda u @ \widetilde{w} \cdot \chi_j$;

(h) since ζ is the ancestor of a path ζ' composed of parts internal to the instance N_w of N and $\mathbf{v}$-cycles over variables occurring free in N_w, and $\xi_i \cdots \xi_j$ is the ancestor of the path $\xi'_i \cdots \xi'_j$ containing ζ', the path $\nu \lambda u @ \widetilde{w}$ is the ancestor of the root edge of N_w, that is, $w = \widetilde{w}$;

(i) ζ is an @-cycle over the application in u;

(j) $\xi_i \cdots \xi_j$ composes of two parts χ_i and χ_j internal to M and a $\mathbf{v}$-cycle $w \lambda u @ \nu \cdot \zeta \cdot \nu \lambda u @ w$ over an occurrence of the variable of u.

The previous procedure can be iterated for the next ξ_k that, after starting in M, goes in N through u; and so on for all the occurrences of u contained in some ξ_i. In this way, we eventually get a sequence of paths internal to M alternated with $\mathbf{v}$-cycles either over x or over variables occurring free in $\lambda x.M$. That is, ψ is an @-cycle.

□

Proposition 6.2.41 *Let* φ *be any legal path of type* @-λ *in* T *but a redex. If* $u = (\lambda x.\, M\ N)$ *is the leftmost-outermost redex traversed by* φ *in* T*, let* $\mathsf{T} \xrightarrow{u} \mathsf{T}'$*. There is a* unique *legal path* φ' *in* T' *whose ancestor is* φ*.*

Proof By hypothesis, $\varphi \neq u$. So, φ does not start or ends at the port of a node in u. Thus, the hypotheses of Lemma 6.2.39 hold and φ is ancestor of a unique wbp φ' in T'. By Lemma 6.2.40, the ancestor ψ of any @-cycle ψ' in φ' is an @-cycle. Thus, by hypothesis, in φ there is a v-cycle $\xi = \nu\, \lambda\, (\varphi_1)^r\, @\, \psi\, @\, \varphi_1\, \lambda\nu$. Correspondingly (note that ξ cannot loop over an occurrence of the variable of u), ψ' contains a subpath $\xi' = \nu'\, \lambda\, (\varphi_1')^r\, @\, \psi'\, @\, \varphi_1''\, \lambda\nu''$, where φ_1 is an ancestor of φ_1' and φ_1''. Using the same technique repeatedly used in the previous lemmas, we can then prove that $\varphi_1' = \varphi_1''$ and $\nu' = \nu''$. (We leave the details of the latter proof to the reader as an exercise.) □

Theorem 6.2.42 (from legal paths to degrees) *For any legal path* φ *of type* @-λ *in a term* T*, there exists a degree* ℓ *of a redex originated along some reduction of* T *such that* $\mathsf{path}(\ell) = \varphi$*.*

Proof By Proposition 6.2.41, when a legal path φ is longer than a simple redex, we can "contract" it into another legal legal path φ' by firing the leftmost-outermost redex that φ traverses. Since φ' is strictly shorter than φ, iterating the process we eventually get a reduction ρ such that the contractum of φ is a single redex ν. Furthermore, since labels are preserved by contraction, the path yielded by the degree of ν is exactly φ, that is, $\mathsf{path}(\mathsf{degree}^{\rho}_{\mathsf{T}}(\nu)) = \varphi$. □

Proposition 6.2.43 (canonical derivation of a legal path) *Let* φ *be any legal path of type* @-λ *but a redex. Assuming that the canonical derivation of a redex is the empty one, the* canonical derivation *of* φ *is obtained by firing the leftmost-outermost redex in* φ*, and iterating this process over its unique residual.*

Proof By induction on the length φ. The base case is obvious. Then, let us take a legal path φ, with $\varphi > 1$. By Lemma 6.2.11, the leftmost-outermost redex traversed by φ is the first-one fired by the canonical derivation yielding the label of φ. Let it be u. By Proposition 6.2.41 the residual of φ w.r.t. u is unique and it is legal. Let it be φ' (notice that its length is shorter than φ). By induction, φ' has a canonical

derivation fitting with the statement. Let it be σ. The derivation $\mathfrak{u}\sigma$ is canonical, by the definition of the extraction relation, and the fact that $\mathfrak{u}$ is needed. Since the derivation $\mathfrak{u}\sigma$ preserves the label of (all the residuals of) φ, the proposition is proved.
To conclude, let us remind that the canonical derivation yielding a redex labeled as φ is unique, for the bijective correspondence between canonical derivations and degrees. □

The previous proposition essentially states the intuitive fact that *all and only* the ("virtual") redexes along a legal path are needed for its contraction to a single redex. Moreover, the uniqueness of the residual of a legal path with respect to its leftmost-outermost redex also provides a unique, standard (and thus canonical) way to perform this contraction.

6.2.6 Legal paths and optimal reductions

Due to Theorem 6.2.42, an optimal implementation of the λ-calculus must always keep a unique representation of every legal path. In particular, legal paths must be *physically* shared during the computation. The optimal evaluation can thus proceed by contracting a legal path, only when there is some mechanism ensuring that every contractum of any legal subpath has a unique representation.

More precisely, suppose to have an explicit operator of duplication (a *fan*, in Lamping's terminology). Consider a legal path φ. An essential condition to get optimality is that a *fan* external to φ can never enter the path. On the other side, a *fan* already internal to φ can be freely moved along φ (provided it does not enter subpaths of type @-λ). That is, we may pursue the duplication *inside* a path @-λ, for the portion of paths we are duplicating eventually belong to different legal paths. As a matter of fact, a *prerequisite chain* for an @-node is always a prefix of a legal path, up to the first atomic edge representing a redex. The interesting fact is that this prefix is always *unique* for every legal path starting from a same node. In a sense, prerequisite chains are the deterministic part of legal paths. Similarly, a prerequisite chain of a λ-node is the *unique* common suffix of every legal path starting from the last edge representing a redex.

6.3 Consistent Paths

Consistent paths were introduced in [GAL92a] as a natural extension of Lamping's proper paths.

Proper paths have been used in proving correctness of Lamping's algorithm. In fact, in order to establish a correct matching relation between fans, we introduced a new data structure, the so-called *contexts*. Contexts store the branching information corresponding to traverse fan-in's and organize it into levels, reflecting in this the stratified organization of terms. In particular, traversing a fan-in with index n, a marker is recorded (pushed) in the n-th level of the context. Markers pushed into a context while traversing fan-in's are then used to decide how to cross fan-out's. Namely, traversing a fan-out with index n, the marker of the port to pass through must match the type of the top marker in the n-th level of the context; at the same time, this traversing causes the consumption of that marker, which is thus popped from the context. Croissants and brackets control instead the structure of contexts, for they modify contexts according to the reorganization of levels they induce on terms: a croissant with index n adds a new level at height n and increases by 1 the height of all the levels above n; a bracket with index n "compresses" and stores the information of level $n+1$ into level n, decreasing by 1 the height of all the levels above n.

6.3.1 Reminder on proper paths

Before to go on, let us recall the definition of proper path (see Section 3.4).

A *level* is an element of the set inductively generated by the following grammar:

(i) $\square$ is a level (the *empty* level);
(ii) if a is a level, then so are $\circ \cdot a$, $* \cdot a$;
(iii) if a and b are levels, then also $\langle a, b\rangle$ is a level.

A *context* is an infinite list of levels containing only a finite number of non-empty levels. Namely:

(i) the *empty* context $\varnothing$ is the infinite list of empty levels, i.e., $\varnothing = \langle \varnothing, \square\rangle$;
(ii) if C is a context and a_0 is a level, then $\langle C, a_0\rangle$ is a context.

Any context A has the following shape:

$$A = \langle \cdots \langle B, a_{n-1}\rangle \cdots, a_0\rangle$$

where $a_0, \ldots, a_{n-1}$ are levels, and B is a context. We will say that $A(i) = a_i$ is the i-th level of the context A, and that B is the subcontext

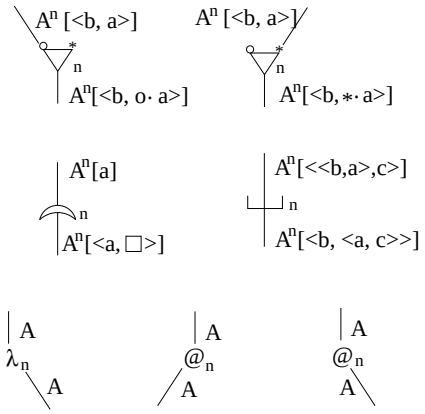
of A at *width* n. We will also say that $A^n[\bullet] = \langle\langle\cdots\langle\bullet, a_{n-1}\rangle,\cdots, a_1\rangle, a_0\rangle$ are the levels of A lower than n, and we will denote by $A^n[C]$ the context obtained replacing C for the subcontext of A at width n (in our case replacing C for B).

Let us also remind that, in spite of their formalization as infinite lists, contexts are indeed finite objects, for in any context only a finite number of levels may be non-empty.

A *proper path* in a sharing graph G is a path such that:

(i) Every edge of the path is labeled with a context.

(ii) Consecutive pairs of edges satisfy one of the following constraints:



Proper paths are identified up to contexts. That is, two proper paths having pairwise equal edges are considered equal, even if their contexts differ.

The problem of proper paths is that they do not exploit the symmetry between β-rule and fan-annihilation. More precisely, we already observed that in optimal reduction there is an operational analogy between these two kind of rules. Such a correspondence is at the base of the definition of *legal path*. In fact, hidden behind the construction of legal paths there is the idea that an @-node is a fan with the principal port oriented towards the function edge and that a λ-node is a fan with the principal port oriented towards the context edge. According to this interpretation, it is immediate to see that β-rule becomes indeed a particular case of fan-annihilation. Nevertheless, proper paths have the following restrictions:

- A path cannot traverse a λ from the binding port to the context port (or vice versa).

- A path may traverse an application from an auxiliary port to the other auxiliary port (i.e., from the context to the argument or vice versa). On the other side, a path cannot traverse an application from its principal port (the function edge) to the auxiliary port corresponding to the argument.

Removing such differences between fans and proper nodes we get the notion of consistent path.

6.3.2 Consistency

For the sake of clarity, with respect to the presentation given in [GAL92a], we shall use different markers for proper nodes and fans. We will thus have the usual symbols $\circ$ and $\star$ for auxiliary ports of fans, and the new symbols # and \$ for auxiliary ports of proper nodes. According to this, the grammar for levels becomes:

$$\mathfrak{a} ::= \Box \mid \circ \cdot \mathfrak{a} \mid \star \cdot \mathfrak{a} \mid \# \cdot \mathfrak{a} \mid \$ \cdot \mathfrak{a} \mid \langle \mathfrak{a}, \mathfrak{b} \rangle$$

In the following, we will assume that contexts are defined in the usual way with the previous extension on the grammar of levels.

Definition 6.3.1 (consistent paths) A *consistent path* in a sharing graph is an undirected path that can be consistently labeled by contexts according to the rules in Figure 6.11.

As for proper paths, consistent labeling of a path is just a sort of correctness criteria. So, paths are equated modulo contexts. Nevertheless, let us remark that the idea behind consistent paths is to find a suitable denotational interpretation of paths, that is, some property invariant under reduction. From this point of view, the denotation associated to a path is the context transformation mapping the input context of the path to the output one. Proper paths yielded this property with respect to rules involving control nodes, hence correctness of Lamping's algorithm has been proved with respect to the λ-term matching the graph (we will come back to this technique in Chapter 7). Nevertheless, β-rule causes a global rearrangement of proper paths. Thus, any denotational property of them is immediately lost in presence of β-reduction. In the case of consistent paths we rather have that any reduction rule—β included—only causes a local rearrangement of the corresponding paths (this should not be surprising, for proper nodes become special cases of fans). Hence, assuming that the initial and final nodes of a consistent

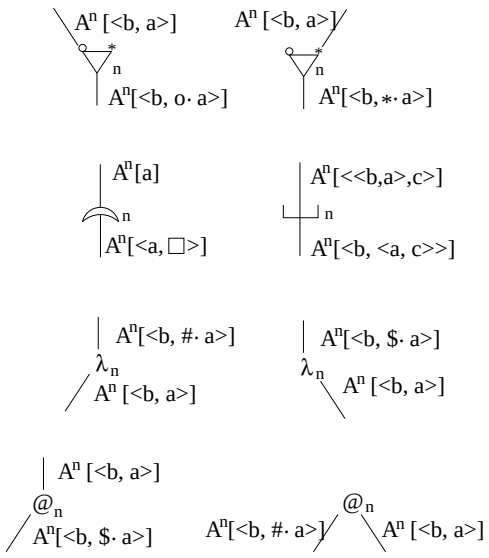


Fig. 6.11. Gonthier's context transformations

path do not disappear along the reduction, the input-output transformation associated to such a path can be seen as its denotation.

Proposition 6.3.2 *The context transformation between the initial and final point of a consistent path is invariant during optimal reduction of the graph (provide that the initial and final nodes of the path have not been erased along the reduction).*

Proof By inspection of the graph rewriting rules. □

The next example shows in details the shape of consistent paths in sharing graphs.

Example 6.3.3 The dotted path in Figure 6.12 is a consistent path. A possible choice of contexts labeling the path is the following one (starting

from the root of the term):

$$\langle \mathsf{X}, \$ \cdot \Box \rangle$$
$$\langle \mathsf{X}, \$ \cdot \$ \cdot \Box \rangle$$
$$\langle \mathsf{X}, \$ \cdot \Box \rangle$$
$$\langle \mathsf{X}, \Box \rangle$$
$$\langle \mathsf{X}, \$ \cdot \Box \rangle$$
$$\langle \langle \mathsf{X}, \$ \cdot \Box \rangle, \Box \rangle$$
$$\langle \langle \mathsf{X}, \$ \cdot \Box \rangle, \circ \cdot \Box \rangle$$
$$\langle \langle \mathsf{X}, \$ \cdot \Box \rangle, \# \cdot \circ \cdot \Box \rangle$$
$$\langle \langle \mathsf{X}, \$ \cdot \Box \rangle, \circ \cdot \Box \rangle$$
$$\langle \langle \mathsf{X}, \Box \rangle, \circ \cdot \Box \rangle$$
$$\langle \langle \langle \mathsf{X}, \Box \rangle, \Box \rangle, \circ \cdot \Box \rangle$$
$$\langle \langle \langle \mathsf{X}, \Box \rangle, \# \cdot \Box \rangle, \circ \cdot \Box \rangle$$
$$\langle \langle \langle \mathsf{X}, \Box \rangle, \# \cdot \Box \rangle, \# \cdot \circ \cdot \Box \rangle$$
$$\langle \langle \langle \mathsf{X}, \Box \rangle, \# \cdot \Box \rangle, \circ \cdot \Box \rangle$$
$$\langle \langle \langle \mathsf{X}, \Box \rangle, \# \cdot \Box \rangle, \Box \rangle$$
$$\langle \langle \mathsf{X}, \Box \rangle, \# \cdot \Box \rangle$$
$$\langle \langle \mathsf{X}, \Box \rangle, \Box \rangle$$

where X is an arbitrary context.

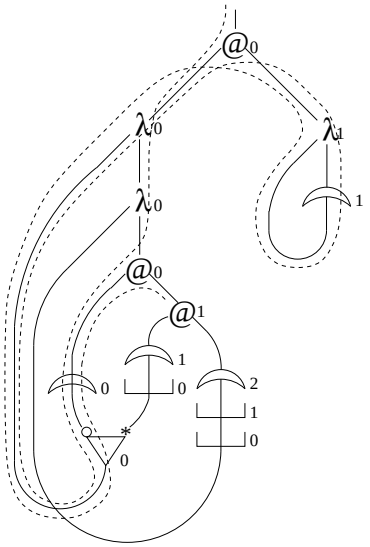


Fig. 6.12. Consistent paths.

Exercise 6.3.4 Verify Proposition 6.3.2 reducing the sharing graph in Example 6.3.3 (Figure 6.12).

Let us note that, constructing the path in Figure 6.12, we could not start with an arbitrary context. We should rather use something like $\langle\langle \cdot, \$ \cdot \mathfrak{b}\rangle \mathfrak{a}\rangle$. This was not the case for proper paths, where starting at the root of a sharing graph we could use any context. Thus, the transformation issued by consistent paths is a relation or, more precisely, a partial function. We should now observe that, building a given path, the shape that we must assign at the root edge is a sort of "observation" of the proper nodes crossed by the path (control nodes do not influence this initial value). Because of this, there is indeed good evidence that we could retrieve a denotation of the term represented by a sharing graph from the set of its consistent paths—in particular, that we could build its Böhm's tree. Anyhow, at present, no formal proof of this fact exists in literature; although Gonthier, Abadi and Lévy claimed it in [GAL92a]. In fact, the sketch of proof of correctness in [GAL92a] rested on that correspondence between consistent paths and Böhm trees. Nevertheless, a detailed proof attempt based on that approach is actually much more complicated than the one followed in the book, for using consistent paths no simple read-back procedure can be provided.

Let us finally remark that the relation between Lamping's and Gonthier's context semantics does rely on the so-called *transparency* property (see Proposition 3.5.10). Namely, if we describe a loop inside the function part of an application, and this function is involved in a β-redex, the context transformation defined by such a loop is inessential to the read-back of the argument part of this application. So, instead of crossing the loop, we can just pass from the context ports of the application to its argument port, as Lamping actually does.

6.4 Regular Paths

Once having understood the notion of consistent path, the reader should not have much problems to shift to *regular paths*. The general idea is the following: instead of considering the nodes of the sharing graph as context transformers, let us directly label each (oriented) edge of the graph with a "weight" in a suitable algebra (the *dynamic algebra*) that essentially embodies the context transformation given by its traversal. The weight of a path is then defined as the (antimorphical) composition of the weights of its edges. Equations of the dynamic algebra will reflect

the consistency conditions of the previous section. In particular, since inconsistency will correspond to have a weight equal to 0, a path will be consistent, or *regular*, when its weight will differ from 0.

This introduction to regular paths is completely apocryphal. As a matter of fact, regular paths have been introduced by Danos and Regnier some time before consistent paths, and with totally different aims (actually, they are the *first* notion of a path ever introduced in the literature, apart from the very restricted case of Lamping's paths).

More precisely, regular paths were devised in the context of Girard's Geometry of Interaction† with the aim of providing a mathematical account of β-reduction (and more generally, of cut elimination) that could describe its apparently global and sequential nature as a sequence of local and parallel elementary moves. To this aim, Danos and Regnier reinterpreted Girard's *Execution Formula* EX as the computation of suitable paths, called *regular*, in the graphical representation of λ-terms (that, as we have seen, is inherited from proof nets of Linear Logic). EX is a scalar value attached to the graph and belonging to the dynamic algebra $\mathcal{L}^*$, an inverse semigroup (see [CP61]) of partial one-to-one transformations of any countable set (say $\mathbb{N}$). The computation of EX (virtual reduction) can be described by a single completely *local and asynchronous* rule: the composition of two edges.

6.4.1 The Dynamic Algebra $\mathcal{LS}$

We give a presentation of the dynamic algebra $\mathcal{LS}$ as an equational theory. We define at the same time the language and the (equational) axioms of $\mathcal{LS}$. Items are:

- a composition function which is associative;
- a neutral constant 1 and an absorbing constant 0 for composition;
- an involution operation u^* satisfying

$$(PQ)^* = Q^*P^* \qquad 0^* = 0 \qquad 1^* = 1$$

  for any P and Q in $\mathcal{LS}$;
- two *multiplicative coefficients* p and q satisfying the *annihilation axioms*:

$$p^*p = q^*q =, 1 \qquad\qquad p^*q = q^*p = 0$$

† "Interaction" means dealing with local computations of a logical nature; "geometry of" means by geometric (or algebraic) tools.

- four *exponential constants* r, s, t, d satisfying the annihilation axioms:

$$\mathsf{r}^*\mathsf{r} = \mathsf{s}^*\mathsf{s} = \mathsf{d}^*\mathsf{d} = \mathsf{t}^*\mathsf{t} = 1 \qquad \mathsf{s}^*\mathsf{r} = 0$$

- a morphism $!$ for composition, 0, 1 and $*$;
- the exponential constants and the morphism $!$ are related by the following *commutation axioms*

$$!(\mathsf{P})\mathsf{r} = \mathsf{r}!(\mathsf{P}) \qquad !(\mathsf{P})\mathsf{s} = \mathsf{s}!(\mathsf{P}) \qquad !(\mathsf{P})\mathsf{t} = \mathsf{t}!^2(\mathsf{P}) \qquad !(\mathsf{P})\mathsf{d} = \mathsf{d}\mathsf{P}.$$

  where P is any element of $\mathcal{LS}$.

Let us now consider an oriented graph labeled by elements of the dynamic algebra. The *weight* of an edge is just the label of the edge. The weight $w(\varphi)$ of a path φ is inductively given by the following rules:

(i) If φ is a null path (a path starting and ending in the same node, crossing no edge), then its weight is 1.
(ii) If φ is $\mathfrak{u}\varphi'$ (resp. $\mathfrak{u}^r\varphi'$), then $w(\varphi) = w(\varphi')w(\mathfrak{u})$ (resp. $w(\varphi')w(\mathfrak{u})^*$), where $\mathfrak{u}$ is an oriented edge.

Note that weights are composed antimorphically with respect to path composition and that we have $w(\varphi^r) = w(\varphi)^*$.

Definition 6.4.1 A path φ is *regular* if and only if $\mathcal{LS} \not\vdash \mathtt{w}(\varphi) = 0$.

Read sequences of $!$ as a denotation for levels, there is an obvious analogy between the operational behavior of d and t as expressed by the commutation rules, and the rewriting rules for croissant and bracket in Lamping's algorithm. Similarly for annihilation and β-rule, in which r and s should be interpreted as the two branches of fans, and p and q as the two branches of application and λ-nodes. Starting from this analogy, the labeling of graphs for λ-terms depicted in Figure 6.13 should come as no surprise (the interesting fact is that this encoding was known *before* Lamping's algorithm).

At first sight the definition of regularity might look pretty ineffective, for it is usually hard to show that something is not provable in an axiomatic theory. Nevertheless, the following theorem due to Regnier shows that in $\mathcal{LS}$ it is rather easy to compute whether a weight of a path is (provably) null or not.

Theorem 6.4.2 (AB^*) *Let* M *be a term and* φ *be a path in* $\mathcal{G}(\mathsf{M})$. *The weight* $w(\varphi)$ *rewrites by the left to right oriented equations of* $\mathcal{LS}$ *either into* 0, *or into an element* AB^* *of* $\mathcal{LS}$, *where* A *and* B *are* positive, *i.e.,*

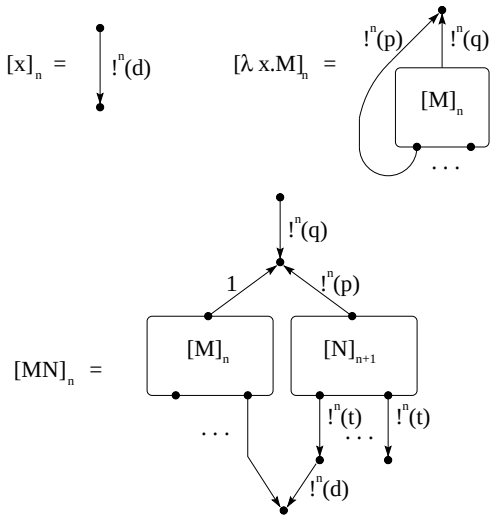


Fig. 6.13. λ-term labeling in the dynamic algebra $\mathcal{LS}$.

an element of $\mathcal{LS}$ possibly equal to 1 and containing neither the constant 0, nor the symbol $*$.

Proof See [Reg92]. □

Let us now suppose that $AB^* = 0$, for some positive A and B. By induction on the number of constants in A, we see that $A^*A = 1$. Similarly, we also have $B^*B = 1$. Thus, we would get $\mathcal{LS} \vdash 0 = 1$. But this is contradicted by the fact that $\mathcal{LS}$ has at least a non trivial model (see section 6.4.2). Thus rewriting a weight into AB^* form shows indeed that it is not provably null in $\mathcal{LS}$.

The foregoing argument has an interesting corollary concerning models of $\mathcal{LS}$.

Corollary 6.4.3 *Let $\mathcal{M}$ be a non trivial model of $\mathcal{LS}$ and M be a term. Then, for every path φ in $\mathcal{G}(M)$ we have*

$$\mathcal{LS} \vdash w(\varphi) = 0 \quad \textit{iff} \quad \mathcal{M} \models w(\varphi) = 0.$$

Remark 6.4.4 The previous corollary is not the case in general: usually a model satisfies more equations than the theory. For instance, one may find some non trivial model of $\mathcal{LS}$ in which $t^*d = 0$, that is not provable in $\mathcal{LS}$ (i.e., there are some other models in which $t^*d \neq 0$). In fact, AB^*

theorem and its corollary are only valid for weights of paths. However, it is a strong result, since it states that any non trivial model of $\mathcal{LS}$ is as good as the theory for deciding regularity.

Example 6.4.5 Let us consider the regular path drawn by a dotted line in Figure 6.14 (let us notice too that this is the consistent one already considered in Figure 6.12). The weight of this path is

$$w = \mathrm{p}^*\mathrm{1}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{1p!(p)!(d)(!(q))}^*\mathrm{p}^*\mathrm{1prd1qq}^*\mathrm{q}^*\mathrm{1q}$$

Let us simplify it (we already erased all the occurrences of 1):

$$\begin{aligned}
w &= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)!(d)(!(q))}^*\mathrm{p}^*\mathrm{prdqq}^*\mathrm{q}^*\mathrm{q} \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)!(d)(!(q))}^*\mathrm{p}^*\mathrm{prdqq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)!(d)(!(q))}^*\mathrm{rdqq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)!(d)!(q}^*\mathrm{)rdqq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)!(d)r!(q}^*\mathrm{)dqq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)!(d)rdq}^*\mathrm{qq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)!(d)rdq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)r!(d)dq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{p!(p)rddq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{pr!(p)ddq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{p}^*\mathrm{prdpdq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{r}^*\mathrm{rdpdq}^* \\
&= \mathrm{p}^*\mathrm{d}^*\mathrm{dpdq}^* \\
&= \mathrm{p}^*\mathrm{pdq}^* \\
&= \mathrm{dq}^*
\end{aligned}$$

The theorem AB^* has a natural counterpart in sharing graphs.

Theorem 6.4.6 *Let* G *be a sharing graph and* ϕ *be a consistent path in* G*. If we reduce* all *redexes internal to* ϕ *(i.e., we normalize it), the residual* ϕ' *of* ϕ *is the composition of two paths* $\phi_1\phi_2$*, where* ϕ_1 *traverses nodes from principal to auxiliary ports, and* ϕ_2 *traverses nodes from auxiliary to principal ports.*

Proof [Sketch] Suppose otherwise. Then we should have inside ϕ' a pair of nodes connected to their principal ports. This is either a redex,

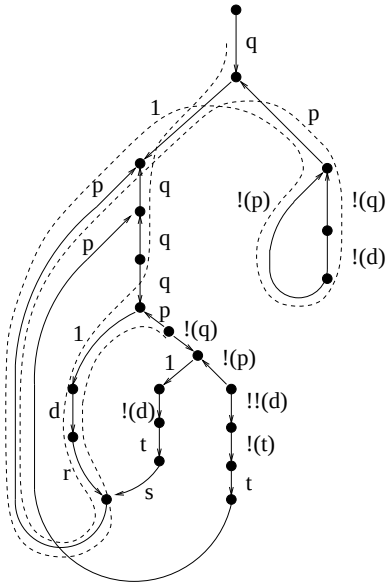


Fig. 6.14. A regular path.

contradicting the hypothesis that ϕ was in normal form, or a deadlock, contradicting the hypothesis that ϕ (and thus ϕ') was consistent. □

For instance, let ϕ be the path in Figure 6.14. If you reduce all redexes internal to ϕ and consider the residual ϕ' of ϕ in the final graph, ϕ' traverses in order a λ of index 0 from the principal to the body port, and a croissant of index 0 from the auxiliary to the principal port. Note the correspondence with the reduced weight of ϕ.

6.4.2 A model

A simple model of the dynamic algebra is given by the the inverse semigroup $\mathcal{L}^*$ of *partial* one-to-one transformations of $\mathbb{N}$ (see [CP61]).

Let 0 and 1 be respectively the nowhere defined transformation and the identity of $\mathbb{N}$. For any element x in $\mathcal{L}^*$, let x^* be its inverse transformation. We have:

$$(xy)z = 7x(yz) \tag{6.0}$$

$$(x^*)^* = x \tag{6.1}$$

$$(xy)^* = y^*x^* \tag{6.2}$$

$$xx^*x = x \tag{6.3}$$

Let $\langle x\rangle = xx^*$ (resp. $\langle x^*\rangle = x^*x$), that is 1 restricted to the codomain (resp. the domain) of x. Obviously, $\langle x\rangle = xx^*$ and $\langle x^*\rangle = x^*x$ are idempotents, since $\langle x\rangle\langle x\rangle = xx^*xx^* = xx^* = \langle x\rangle$, and symmetrically for $\langle x^*\rangle$.

The very basic property of inverse semigroups is that idempotents commute. Namely:

$$\langle x\rangle\langle y\rangle = \langle y\rangle\langle x\rangle \tag{6.4}$$

Axioms (0)-(4) provide a possible axiomatization of inverse semigroups (see [Pet84]).

Let us now consider some interesting consequences of these axioms.

Proposition 6.4.7 *Let a be an idempotent. Then, $a = a^*$.*

Proof Let $a = aa$, $a^* = a^*a^*$. We proving that $a = aa^*$.

$$\begin{aligned} a &= aa^*a \\ &= aaa^*a^*a \\ &= a\langle a\rangle\langle a^*\rangle \\ &= a\langle a^*\rangle\langle a\rangle \\ &= aa^*aaa^* \\ &= aaa^* \\ &= aa^* \end{aligned}$$

Then, $a^* = (aa^*)^* = (a^*)^*a^* = aa^*$, and by transitivity $a = a^*$. □

Corollary 6.4.8 *If a and b are idempotents, then $ab = ba$.*

Proof Obvious, since by the previous proposition $a = \langle a\rangle$. □

We already have the basic ingredients for interpreting the monomials of the dynamic algebra $\mathcal{LS}$. In particular, we can take:

$$\begin{aligned} [\![1]\!]_{\mathcal{L}^*} &= 1 \\ [\![0]\!]_{\mathcal{L}^*} &= 0 \\ [\![(a)^*]\!]_{\mathcal{L}^*} &= ([\![a]\!]_{\mathcal{L}^*})^* \\ [\![ab]\!]_{\mathcal{L}^*} &= [\![a]\!]_{\mathcal{L}^*}[\![b]\!]_{\mathcal{L}^*} \end{aligned}$$

Our next step is to give an interpretation to $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \mathsf{d}, \mathsf{t}$ and the "of course" symbol.

The constants p and q are not a problem: just take any two *total* transformations with disjoint codomains. For instance:

$$\begin{aligned} [\![\mathsf{p}]\!]_{\mathcal{L}^*}(n) &= 2n \\ [\![\mathsf{q}]\!]_{\mathcal{L}^*}(n) &= 2n+1 \end{aligned}$$

Now, let $(m.n) \mapsto \lceil m, n \rceil$ be a one-to-one mapping of $\mathbb{N}^2$ onto $\mathbb{N}$. We shall denote by $\lceil n_1, \ldots, n_k \rceil$ the integer $\lceil n_1, \lceil n_2, \ldots \lceil n_{k-1}, n_k \rceil, \ldots \rceil \rceil$. We can define the functional $!_{\mathcal{L}^*} : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$, assuming that:

$$!_{\mathcal{L}^*}(f)\lceil m, n \rceil = \lceil m, f(n) \rceil$$

for any transformation f. Thus, for any monomial a of $\mathcal{LS}$, we can take:

$$[\![!(\mathsf{a})]\!]_{\mathcal{L}^*} = !_{\mathcal{L}^*}([\![\mathsf{a}]\!]_{\mathcal{L}^*})$$

Finally, let k be an arbitrary integer and ρ, σ and τ three *total* transformations such that ρ and σ have disjoint codomains. We define:

$$\begin{aligned} [\![\mathsf{d}]\!]_{\mathcal{L}^*}(n) &= \lceil \mathsf{k}, n \rceil \\ [\![\mathsf{t}]\!]_{\mathcal{L}^*}\lceil l, \lceil m, n \rceil \rceil &= \lceil \tau(\lceil l, m \rceil), n \rceil \\ [\![\mathsf{r}]\!]_{\mathcal{L}^*}\lceil m, n \rceil &= \lceil \rho(m), n \rceil \\ [\![\mathsf{s}]\!]_{\mathcal{L}^*}\lceil m, n \rceil &= \lceil \sigma(m), n \rceil \end{aligned}$$

We leave as an exercise the easy verification that the previous interpretation satisfies the axioms of $\mathcal{LS}$. We only remark that the definition of $[\![\,.\,]\!]_{\mathcal{L}^*}$ has been subject to a lot of arbitrary choices, starting from the bijective function $(m.n) \mapsto \lceil m, n \rceil$. As a consequence, it is possible to build without much effort some other interpretation satisfying additional equations. For instance, $[\![(\mathsf{t}^*\mathsf{d})]\!]_{\mathcal{L}^*} = [\![\mathsf{0}]\!]_{\mathcal{L}^*}$ if and only if k is not in the range of τ.

6.4.3 Virtual Reduction

Let us start adding some more structure to our algebra. In the model of partial transformations $\mathcal{L}^*$, it is natural to introduce a sort of *complementation* of a transformation f, that is an operation defined by $\overline{\mathsf{f}} = \mathsf{1} - \langle \mathsf{f} \rangle = \mathsf{1} - \mathsf{f}\mathsf{f}^*$. The main equations concerning this operation would be the following:

$$\overline{\mathsf{1}} = \mathsf{0} \tag{6.5}$$

$$\overline{0} = 1 \tag{6.6}$$

$$x\overline{y} = \overline{xy}x \tag{6.7}$$

Finally, let us remind that 1 is a neutral element and that 0 is absorbing. Namely, that we have:

$$1x = x = x1 \tag{6.8}$$

$$0x = 0 = x0 \tag{6.9}$$

These axioms, in conjunction with axioms (1)-(4) of the previous section, provide an axiomatization of the notion of *bar-inverse monoid* introduced by Danos and Regnier in [DR93]. Let us see some interesting consequences of them.

Proposition 6.4.9

1) $\overline{x}x = 0$;
2) $\overline{x}\,\overline{x} = \overline{x}$;
3) $\overline{x}^* = \overline{x}$
4) $x^*\overline{x} = 0$;
5) $\overline{x}\,\overline{y} = \overline{y}\,\overline{x}$;
6) $\overline{x}\langle y\rangle = \langle y\rangle\overline{x}$;
7) $\overline{xy}\,\overline{x\overline{y}} = \overline{x}$;
8) $x\overline{\overline{x^*}} = x$;
9) $\overline{\overline{x}}x = x$;
10) $\overline{xx^*} = \overline{x}$.

Proof

1) $\overline{x}x = \overline{x1}x = x\overline{1} = x0 = 0$;
2) $\overline{x}\,\overline{x} = \overline{\overline{x}x}\overline{x} = \overline{0}\,\overline{x} = 1\overline{x} = \overline{x}$;
3) by 2) and Proposition 6.4.7;
4) $x^*\overline{x} = x^*\overline{x}^* = (\overline{x}x)^* = 0^* = 0$;
5) by 2) and Corollary 6.4.8;
6) by 2) and Corollary 6.4.8;
7) $\overline{xy}\,\overline{x\overline{y}} = \overline{x\overline{y}}\,\overline{xy} = \overline{\overline{xy}x}\,\overline{xy} = \overline{xy}\,\overline{x} = \overline{x}\,\overline{xy} = \overline{\overline{x}xy}\,\overline{x} = \overline{0y}\,\overline{x} = 1\overline{x} = \overline{x}$;
8) $x\overline{\overline{x^*}} = x\overline{x^*}x = \overline{0}x = x$;
9) let us prove that $(\overline{\overline{x}}x)^* = x^*$. We have $(\overline{\overline{x}}x)^* = x^*\overline{\overline{x}}^* = x^*\overline{\overline{x}} = x$, by the previous property;
10) $\overline{xx^*} = \overline{xx^*}1 = \overline{xx^*}\,\overline{x\overline{x^*}} = \overline{x}$, by 7).

□

Proposition 6.4.10 *The following conditions are equivalent:*

1) $\langle x\rangle\langle y\rangle = 0$
2) $x^*y = 0$
3) $\overline{y}x = x$
4) $\overline{x}y = y$

Proof

1) $\Rightarrow$ 2) Suppose $xx^*yy^* = 0$. Then:

$$x^*y = x^*xx^*yy^*y = 0$$

2) $\Rightarrow$ 3) Suppose $x^*y = 0$. Then:

$$x^*\overline{y} = \overline{x^*y}x^* = \overline{0}x^* = x^*$$

and thus

$$x = (x^*\overline{y})^* = (\overline{y})^*x = \overline{y}x$$

3) $\Rightarrow$ 4) Suppose $\overline{y}x = x$ and let us prove that $y^*\overline{x} = y^*$. Using the fact that $y^*\overline{y} = 0$, we have:

$$y^*\overline{x} = \overline{y^*x}y^* = \overline{y^*\overline{y}x}y^* = \overline{0}y^* = y^*$$

Then:

$$y = (y^*\overline{x})^* = (\overline{x})^*y = \overline{x}y$$

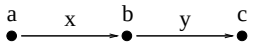
4) $\Rightarrow$ 1) Recall that $x^*\overline{x} = 0$. Then:

$$xx^*yy^* = xx^*\overline{x}yy^* = 0$$

□

Using the previous algebraic material, we can now define an interesting notion of *virtual reduction* on our graphs labeled by weights of the dynamic algebra.

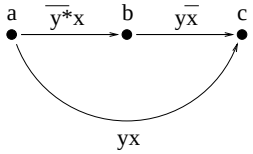
Let us assume to have the following configuration, where we have two edges $a \xrightarrow{x} b$ and $b \xrightarrow{y} c$:



Remark 6.4.11 According to the interpretation in $\mathcal{LS}$, $*$ means to reverse the orientation of a path. Thus, in the sequel, we shall implicitly identify the edges $a \xrightarrow{x} b$ and $b \xrightarrow{x^*} a$.

The weight of the composed path is yx. So, we could consistently add a new edge between a and c with weight yx. We would also be tempted to remove the two original edges, since we already computed their composition. However, this could be wrong, since there could exist other incoming edges in b, and other possible paths which have not been considered yet. On the other side, we cannot just leave the original edges as they are, since they could compose together over and over again, preventing any sensible notion of *reduction* of the graph.

Intuitively, we would just recall that the two edges labeled by x and y have been already composed. An elegant way to do that, is remove from y of the part of its weight fired in the composition with x (and symmetrically for x). Formally, this is achieved restricting y by the complementary of the left unit of y, that is $\overline{y}$ (and symmetrically for x). So, the *virtual reduction* of the above configuration leads to:



If we try to compose again the residual of the original edges, we would now get a new edge with weight 0. Indeed:

$$y\overline{x}\overline{y^*}x = y\overline{y^*}\overline{x}x = 0$$

Obviously, there is no interest in adding 0 weighted edges to the graph, so we can just prevent the previous recomposition by requiring that *the operation of virtual reduction can be only applied to pair of edges whose composed weight is different from* 0.

Actually, the previous operation of reduction can be seen as the composition of two more elementary steps *splitting* and *merging*.

- *Splitting* an edge $a \xrightarrow{x} b$ by a weight w amounts to replace the edge with two edges $a \xrightarrow{wx} b$ and $a \xrightarrow{\overline{w}x} b$.
- The operation of *merging* two *distinct* edges $a \xrightarrow{x} b$ and $b \xrightarrow{y} c$ is only defined if $\langle x \rangle = \langle y^* \rangle$; in this case, it amounts to erasing the two original edges and replacing them with a new edge $a \xrightarrow{xy} c$.

Definition 6.4.12 (virtual reduction) The *reduction* of two distinct edges $a \xrightarrow{x} b$ and $b \xrightarrow{y} c$ is only defined if $xy \neq 0$. In this case, it consists in:

(i) Splitting $a \xrightarrow{x} b$ by $\langle y^* \rangle$, that is, replacing it by $a \xrightarrow{\langle y^* \rangle x} b$ and $a \xrightarrow{\overline{\langle y^* \rangle} x} b$.

(ii) Splitting $c \xrightarrow{y^*} b$ by $\langle x \rangle$, that is, replacing it by $c \xrightarrow{\langle x \rangle y^*} b$ and $c \xrightarrow{\overline{\langle x \rangle} y^*} b$.

(iii) Merging the edges $a \xrightarrow{\langle y^* \rangle x} b$ and $b \xrightarrow{y\langle x \rangle} c$, that is, replacing them with a new edge $a \xrightarrow{w} c$, where $w = y\langle x \rangle\langle y^* \rangle x = y\langle y^* \rangle\langle x \rangle x = yx$.

As we have seen, during the virtual reduction of two edges $a \xrightarrow{x} b$ and $b \xrightarrow{y} c$ their weights are modified in order to prevent their recomposition. However, what about other possible edges colliding in b? (e.g., as in the situation of Figure 6.15) Can we ensure that we do not unadvisedly modify the weight of other paths by our notion of reduction?

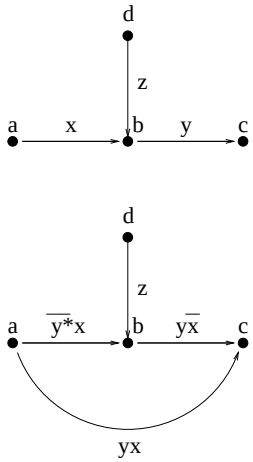


Fig. 6.15. Colliding edges.

The answer, in general, is no! Suppose for instance that in the structure of Figure 6.15 $z = x$: obviously, after reducing $a \xrightarrow{x} b$ and $b \xrightarrow{y} c$ we loose the path from d to c which has now weight 0. However, the idea is that such a configuration is *pathological*. In a sense, if the two weights x and z are not disjoint, they are conflicting on the use of y, and this kind of conflict should be clearly prevented in a "sound" structure. Then, what we may expect, is that the virtual reduction of a "sound" structure should preserve its soundness.

The required notion of soundness is the following:

Definition 6.4.13 Let $a \xrightarrow{x} d$, $b \xrightarrow{y} d$ and $c \xrightarrow{x} d$ three distinct coinci-

dent edges. We say that they are *well split* if and only if:

$$\langle x\rangle\langle y\rangle\langle z\rangle = 0$$

A graph is well split when the well split condition is satisfied by all its edges.

Proposition 6.4.14 *Well splitness is preserved by splitting, merging and (hence) by virtual reduction.*

Proof Exercise. □

We also leave as an exercise to the reader the proof that any graph obtained by lambda terms using the encoding of Figure 6.13 is well split.

The most important property of virtual reduction is that, on a well split graph, it preserves the weights of paths.

Consider again the case of Figure 6.15, and the path leading from d to c In the initial configuration, this path has a weight yz, while after the reduction, its weight is $y\overline{x}z$. We have to prove that they are equal.

Our hypothesis is that the graph is well split, so we have:

$$\langle x\rangle\langle y^*\rangle\langle z\rangle = 0$$

Let us recall that $\langle a\rangle\langle b\rangle = 0$ if and only if $\overline{a}b = b$, and note that $\langle y^*\rangle\langle z\rangle = \langle\langle y^*\rangle\langle z\rangle\rangle$ since it is an idempotent. So, we have:

$$\langle x\rangle\langle y^*\rangle\langle z\rangle = 0$$

if and only if

$$\overline{x}\langle y^*\rangle\langle z\rangle = \langle y^*\rangle\overline{x}\langle z\rangle = \langle y^*\rangle\langle z\rangle$$

and finally

$$y\overline{x}z = y\langle y^*\rangle\overline{x}\langle z\rangle z = y\langle y^*\rangle\langle z\rangle z = yz$$

The proof is similar if we consider the path from d to a (the path from a to c is obviously preserved by reduction).

The reader interested in learning more on virtual reduction and its properties (most remarkably, *confluence*) is referred to [DR93].

6.5 Relating Paths

In this section we will eventually show that all the previous notions of paths coincide. For the sake of clarity, let us remind and fix some notation.

We shall start adding some notation to the labeled graphs, let us call them *dynamic graphs*, already met in studying regular paths. As already notices, although such graphs are unoriented, their edges have a natural orientation corresponding to the arrows in the figures below (see also Figure 6.13).

Each node $\mathtt{m}$ has a *depth* (an *index* in sharing graphs terminology) which is a positive integer marking $\mathtt{m}$. Similarly, an edge is at depth $\mathtt{i}$ if its final node (w.r.t. the natural orientation) is at depth $\mathtt{i}$ and the depth of a path is the smallest depth of the nodes it traverses.

Among the edges incident to a node we always distinguish a set of *premises*, that is, a set of entering edges fixed by the type of the node. Each edge of a dynamic graph is the premise of at most one node (i.e., the natural orientation is not contradictory) and it is labeled by a *weight* that is an element of the dynamic algebra $\mathcal{LS}$. The weight of an edge corresponds to its natural orientation; thus, an important property that will be used in associating weights to paths, is that reversing the direction of an edge correspond to the star operation on weights. (Let us remark that this interpretation is sound because of the previous assumption on premises.)

In addition to the usual nodes we have two *communication* nodes (needed in order to be consistent with the requirements above): *context* ($\mathtt{c}$) and *variable* ($\mathtt{v}$). Hence, summing up, the nodes of a dynamic graph can be of three different kinds:

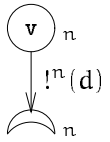
(i) *Communication* nodes: *context* (c) and *variable* (v), with no premises.

(ii) *Logical* nodes: *lambda* and *application* nodes with two associated premises.

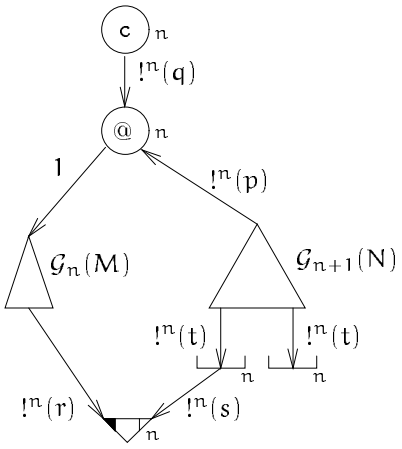
(iii) *Control* nodes: *croissant* and *bracket*, with one premise; *fan* with two premises; *weakening* with no premise.

For each integer $\mathtt{n}$ and each term M we define by induction on M a graph $\mathcal{G}_{\mathtt{n}}(\mathsf{M})$ together with its root node (represented on the figure as the topmost node). The translation is defined so that each free variable of M is associated with a unique control node in $\mathcal{G}(\mathsf{M})$ (at the bottom of the graph) with no exiting edge; these control nodes are the *free nodes* of $\mathcal{G}(\mathsf{M})$. The translation of a lambda-term M is defined to be $\mathcal{G}(\mathsf{M}) = \mathcal{G}_{\mathtt{0}}(\mathsf{M})$.

Variable x.

$\mathcal{G}_n(x) =$ 

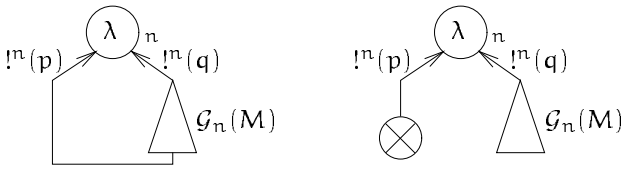
Application (MN).

$\mathcal{G}_n(MN) =$ 

The fan at the bottom of the graph merges two free nodes corresponding to the same variable; at the same time, each pair of corresponding variables occurring free in M and N is linked by means of a fan.
The premises of the application are respectively called the context and the argument edges of the application; the argument node of the application is the initial node of the argument edge. The exiting edge is called the function edge and its final node is the function node of the application.

Abstraction λx.M. There are two cases whether the variable does or does not appear in M. In the former one we link the corresponding node to the lambda node; in the latter one we link a

weakening node to the lambda node:

$\mathcal{G}_n(\lambda x M) =$ 

The premises of the lambda are respectively called the variable and the body edges; the initial node of the body edge is the body node of the abstraction.

If $\mathtt{n}$ is a context, variable or lambda node at depth n in $\mathcal{G}_n(M)$, then the set of nodes below $\mathtt{n}$ at depth greater than n must be of the form $\mathcal{G}_n(N)$, for some subterm N of M. The node $\mathtt{n}$ is the root of N in M; furthermore, if N differs from M, the edge on top of $\mathtt{n}$ is the root edge of N in M or also the root of $\mathtt{n}$.

A path is a sequence of consecutive nodes, or equivalently, a sequence of oriented edges and reversed oriented edges w.r.t. the natural orientation. We shall use both conventions and even mix them without mention. The reverse of a path φ is denoted φ^r. A path is *straight* if it contains no subpath of the form $\varphi\varphi^r$ nor $u v^r$ where u and v are the two premises of some binary node. From now on we assume that all the paths are straight. Note also that all well balanced paths are straight by definition.

The weight of a path is a dynamic graph is the antimorphical composition of the weights of its edges. Namely, the weight of a path $w(\phi)$ is given by the following rule for (antimorphical) composition:

(i) If $\varphi = u$ (i.e., an edge crossed according to its natural orientation), then $w(\phi) = w(u)$ is equal to the weight of u.
(ii) If $\varphi = \psi\psi'$, then $w(\phi) = w(\psi')w(\psi)$.
(iii) If $\varphi = \psi^r$, then $w(\phi) = w(\psi)^*$.

We will say that a path is of type t_1-t_2 if it starts from a node of type t_1 and ends in a node of type t_2. In particular an edge of type @-λ will be called a redex edge. There is a one to one correspondence between redexes in M and redex edges in $\mathcal{G}(M)$. We say that a path φ crosses a redex if it contains the corresponding redex edge.

Beta reduction induces a notion of *residual edge*, defined in the standard way (we already met it several times). This notion can be easily extended to paths in an edge-wise manner. Moreover, let us remark again that paths may have no more than one residual.

6.5.1 Discriminants

To each variable node (occurrence of variable) $\mathtt{v}$ we associate a *discriminant* $\gamma_{\mathtt{v}}$. Namely, the maximal path starting at $\mathtt{v}$ and moving downwards. Let us note that any discriminant is uniquely determined by $\mathtt{v}$ and that it contains control nodes only. The last node of $\gamma_{\mathtt{v}}$ is the *discriminant node* of $\gamma_{\mathtt{v}}$. The discriminant node of a discriminant $\gamma_{\mathtt{v}}$ is obviously connected to a λ-node λ binding $\mathtt{v}$. Thus, we will also say that $\gamma_{\mathtt{v}}$ is bound by this λ.

Let γ be a discriminant. The depth of its variable node is the maximal one in γ, while the depth of its discriminant node $\mathtt{m}$ is the minimal one; in particular, when γ is bound by some λ, the depth of $\mathtt{m}$ is equal to the depth of λ. The difference between these maximal and minimal depths is the *lift* of γ.

Let γ be a discriminant of lift $\mathtt{c}$ in $\mathcal{G}(\mathsf{M})$ such that $\mathtt{k}$ is the depth of its discriminant node. The weight of γ is equal to $!^{\mathtt{k}}(\mathsf{G})$ for some G of the form:

$$\mathsf{G} = \omega_1 \mathtt{t}!(\omega_2 \mathtt{t}!(\ldots \omega_{\mathtt{k}} \mathtt{t}!(\omega_{\mathtt{c}+1}\mathtt{d})))$$

where all the $\omega_{\mathtt{i}}$ are products of $\mathtt{r}$'s and $\mathtt{s}$'s only. More in general, any element of $\mathcal{LS}$ as the G above will be called a *w-discriminant* of lift $\mathtt{c}$.

Any w-discriminant G of lift $\mathtt{c}$ satisfies the following *generalized swap equation*:

$$!(\mathsf{P})\mathsf{G} = \mathsf{G}!^{\mathtt{c}}(\mathsf{P})$$

for any monomial P in $\mathcal{LS}$.

Let γ and γ' be two discriminants bound by the same lambda node λ. If $!^{\mathtt{k}}(\mathsf{G})$ and $!^{\mathtt{k}}(\mathsf{G}')$ are the weights of γ and γ', then the two w-discriminants G and G' satisfies the following *generalized zap equation*:

$$\mathsf{G}^*\mathsf{G}' = \delta_{\gamma\gamma'}$$

where $\delta_{\gamma\gamma'}$ is the Kroenecker function (i.e., $\delta_{\gamma\gamma'} = 1$ when $\gamma = \gamma'$, and $\delta_{\gamma\gamma'} = 0$ otherwise).

6.5.2 Legal paths are regular and vice-versa

We shall prove in this section the coincidence of legal paths and (well-balanced) regular paths.

For the regularity of legal paths we need a couple of lemmas. The first one is a restatement of Proposition 6.2.41 and we recall it here only for the sake of clarity.

Lemma 6.5.1 *Let* M *be a term,* φ *a legal path in* $\mathcal{G}(M)$*,* ρ *the leftmost outermost redex crossed by* φ *and* M' *the term obtained by firing* ρ*. Then* φ *has a* unique *residual* φ' *in* M' *and* φ' *is in turn legal.*

The second one is called *lifting lemma.*

Lemma 6.5.2 (Lifting Lemma) *Let* φ *be a straight path in* $\mathcal{G}(M)$*,* P *be its weight, and* ρ *be the leftmost-outermost redex crossed by* φ*. Suppose* φ *has a residual* φ' *by the reduction of* ρ *and let* P' *be the weight of* φ'*. Then:*

$$P = AP'B^*$$

for some positive A *and* B *in* $\mathcal{LS}$*.*

Proof [Sketch of the proof.] Let @ and λ be the application and the lambda nodes defining the redex ρ, and u be the function edge of @. We use some key properties of φ w.r.t. its leftmost outermost redex. Namely, under the hypotheses of the lemma, we use the fact that the path φ may be decomposed as

$$\varphi = (\varphi_0\,\gamma_0\,v\,u\,v'^*)\,\psi_0\,v'\,u\,v^*\,\gamma_0{}^*\,\varphi_1 \cdots \varphi_n\gamma_n\,v\,u\,v'^*\,\psi_n\,(v'\,u\,v^*\,\gamma_n{}^*\,\varphi_{n+1})$$

where: (i) v and v' are respectively the variable edge of λ and the argument edge of @; (ii) any γ_i is the discriminant of an occurrence of the variable bound by λ; (iii) any φ_i is a subpath of φ lying completely outside the argument of @; (iv) any ψ_i is a subpath of φ lying completely inside the argument of @; (v) the parenthesized subpaths may possibly be empty.
From the decomposition above, a straightforward computation of the weight gives the result. □

Remark 6.5.3 The theorem AB^* is an obvious consequence of this lemma. One has just to prove that if φ has no residual then its weight is null.

Theorem 6.5.4 *Every legal path* φ *is regular.*

Proof We prove by induction on the length of φ that its weight is in the AB^* form. If φ is the function edge of an application node, then its weight is 1; so, it is in AB^* form. Otherwise by definition of wbp, φ must cross some redex. Let ρ be the leftmost outermost redex of φ. If φ' is the residual of φ by the contraction of ρ (by lemma 6.5.1, φ' exists

and is unique), then by induction hypothesis, the weight of φ' is AB^* for some positive elements A and B of $\mathcal{LS}$. But by lifting lemma, there are some positive C and D such that $w(\varphi) = Cw(\varphi')D^* = CAB^*D^*$. □

Unfortunately, the converse of the previous theorem is not as easy. Again, we need a few preliminary results.

Proposition 6.5.5 (Rendez-vous property) *Let P be the weight of a wbp φ linking two nodes $\mathtt{m}$ and $\mathtt{n}$ and let $\mathfrak{m}$ and $\mathfrak{n}$ be the respective depth of $\mathtt{m}$ and $\mathtt{n}$. Then for any element X of $\mathcal{LS}$ we have:*

$$!^{\mathfrak{n}}(X)P = P!^{\mathfrak{m}}(X)$$

Proof When $P = 0$ the proposition is obvious. So, let us assume that P is not null. The proof is by induction on the structure of φ.

- φ *is an edge*: φ is the function edge of some application node; so, $P = 1$ and $\mathfrak{n} = \mathfrak{m}$.
- $\varphi = \varphi_1 \mathfrak{u} \varphi_2 \mathfrak{u}'^{\mathfrak{r}}$: By hypothesis, both φ_1 and φ_2 are wbp's. Let P_1 and P_2 be their respective weights. Let us note that the ending node of φ_1 and the initial node of φ_2 are at the same depth, say k; moreover, the final node of φ_2 has the same depth $\mathfrak{n}$ of the node $\mathtt{n}$, for $\mathfrak{u}$ and $\mathfrak{u}'$ are respectively context and body edges. Then, $P = !^{\mathfrak{n}}(q^*)P_2!^{k}(q)P_1 = P_2P_1$, by induction hypothesis and the annihilation equations of q. Now, using the induction hypothesis on φ_1 and φ_2, we conclude by:
$$!^{\mathfrak{n}}(X)P = !^{\mathfrak{n}}(X)P_2P_1 = P_2!^{k}(X)P_1 = P_2P_1!^{\mathfrak{m}}(X) = P!^{\mathfrak{m}}(X).$$
- $\varphi = \varphi_1 \gamma \mathfrak{u} (\varphi_2)^{\mathfrak{r}} \mathfrak{u}'^{\mathfrak{r}}$: The computation is essentially similar to the one in the previous case. The only difference is that we have to use the swapping equation of the discriminant crossed by φ.

□

Lemma 6.5.6 (@-cycle property) *Let @ be an application node in $\mathcal{G}(M)$ at depth k, $\mathfrak{u}$ be its application edge and $\mathfrak{u}^{\mathfrak{r}}\psi\mathfrak{u}$ be an @-cycle starting at @. There are two monomials U and π such that:*

$$w(\psi) = !^{k+1}(U)\pi$$

where π is a k-swapping idempotent, i.e., π satisfies the two equations:

$$\begin{aligned} \pi\pi &= \pi \\ !^{k}(X)\pi &= \pi!^{k}(X) \end{aligned}$$

for any monomial X.

Proof If the weight of ψ is null, then the proposition is true for $U = \pi = 0$. So, let us suppose that $w(\psi) \neq 0$.
Let N be the argument of the application @. By definition of @-cycle, ψ has the following form:

$$\psi_0 v_1 \gamma_1 \nu_1 \lambda_1 \varphi^r @_1 u_1^r \phi_1 u_1 @_1 \varphi \lambda_1 \nu_1^r \gamma_1^r v_1 \psi_1 \dots \psi_n$$

where, for each i, ψ_i is internal to N, γ_i is a discriminant for some free variable v_i of N, λ_i is its binder (external to N), φ_i is a virtual redex between $@_i$ e λ_i, u_i is the argument edge of $@_i$, and $u_i^r \phi_i u_i$ is an @-cycle.
We prove by induction on n, and then on ψ, that any path of this form has the @-cycle property (note that such a path might not be a cycle).
Let $n = 0$. In this case, $\psi = \psi_0$ is entirely contained in N. Then, its weight has the form $w(\psi) = !^{k+1}(X)$, for some X, and the proposition is satisfied by taking $\pi = 1$.
If $n > 0$, let l_1 be the depth of λ_1. The weight of γ_1 is of the form $!^{l_1}(G_1)$ for some w-discriminant G_1 of lift c_1. Furthermore, let k_1 be the depth of $@_1$. Since $u_1^r \phi_1 u_1$ is a @-cycle, by induction hypothesis (on the length of the cycle ϕ_1) its weight is of the form

$$w(\phi_1) = !^{k_1+1}(V_1)\pi_1$$

for some monomial V_1 and some k_1-swapping idempotent π_1. Moreover, the suffix $\psi_1 \dots \psi_n$, although it is not a @-cycle, still has the right shape. Thus, by induction hypothesis (on the number of components n in the decomposition at the beginning of the proof) its weight must be

$$w(\psi_1 \dots \psi_n) = !^{k+1}(V)\pi$$

for some monomial V and some k-swapping idempotent π. Summing up, if $w(\psi_1) = U_1$, the weight of ψ is:

$$w(\psi) = !^{k+1}(V)\pi !^{l_1}(G_1^*) !^{l_1}(p^*) U_1 !^{k_1}(p) !^{k_1+1}(V_1)\pi_1 !^{k_1}(p^*) U_1^* !^{l_1}(p) !^{l_1}(G_1) !^{k+1}(W_0)$$

Since φ_1 is a wbp linking two nodes whose respective depths are k_1 and l_1, by the rendez-vous property, we have:

$$!^{l_1}(X)U_1 = U_1 !^{k_1}(X)$$

for any monomial X. In particular:

$$!^{l_1}(p^*)U_1 !^{k_1}(p) = !^{l_1}(p^*) !^{l_1}(p) U_1 = U_1$$

The weight of ψ can then be simplified to:

$$w(\psi) = !^{k+1}(V)\pi!^{l_1}(G_1^*)U_1!^{k_1+1}(V_1)\pi_1 U_1^*!^{l_1}(G_1)!^{k+1}(W_0)$$

Let $X = !^{l_1}(G_1^*)U_1!^{k_1+1}(V_1)\pi_1 U_1^*!^{l_1}(G_1)$. We have that:

$$\begin{array}{rcll}
X & = & !^{l_1}(G_1^*)U_1!^{k_1+1}(V_1)\pi_1 U_1^*!^{l_1}(G_1) & \\
 & = & U_1!^{k_1}(G_1^*)!^{k_1+1}(V_1)\pi_1 U_1^*!^{l_1}(G_1) & \text{rendez-vous for } U_1 \\
 & = & U_1!^{k_1}(G_1^*!(V_1))\pi_1 U_1^*!^{l_1}(G_1) & \\
 & = & U_1!^{k_1}(!^{c_1}(V_1)G_1^*)\pi_1 U_1^*!^{l_1}(G_1) & \text{swap-equation for } G_1^* \\
 & = & U_1!^{k_1+c_1}(V_1)!^{k_1}(G_1^*)\pi_1 U_1^*!^{l_1}(G_1) & \\
 & = & U_1!^{k_1+c_1}(V_1)\pi_1!^{k_1}(G_1^*)U_1^*!^{l_1}(G_1) & \text{swap-equation for } \pi_1 \\
 & = & U_1!^{k_1+c_1}(V_1)\pi_1 U_1^*!^{l_1}(G_1^*)!^{l_1}(G_1) & \text{rendez-vous for } U_1 \\
 & = & U_1!^{k_1+c_1}(V_1)\pi_1 U_1^* & \\
 & = & !^{l_1+c_1}(V_1)U_1\pi_1 U_1^* &
\end{array}$$

Thus, we get:

$$w(\psi) = !^{k+1}(V)\pi!^{l_1+c_1}(V_1)U_1\pi_1 U_1^*!^{k+1}(W_0)$$

Since l_1 is the depth of λ_1, it is also the depth of the discriminant node of γ_1. Thus $l_1 \leq k$. That is, there exists a $d_1 \geq 0$ such that $k = l_1 + d_1$. Now, let Y be any monomial.

$$\begin{array}{rcll}
U_1\pi_1 U_1^*!^k(Y) & = & U_1\pi_1 U_1^*!^{l_1+d_1}(Y) & \\
 & = & U_1\pi_1!^{k_1+d_1}(Y)U_1^* & \text{rendez-vous property for } U_1^* \\
 & = & U_1!^{k_1+d_1}(Y)\pi_1 U_1^* & \text{swapping property for } \pi_1 \\
 & = & !^{l_1+d_1}(Y)U_1\pi_1 U_1^* & \text{rendez-vous property for } U_1 \\
 & = & !^k(Y)U_1\pi_1 U_1^* &
\end{array}$$

So, $U_1\pi_1 U_1^*$ is k-swapping. Using the permutation property for idempotents, it is also easy to prove that $U_1\pi_1 U_1^*$ is idempotent.
Let $\pi_1' = U_1\pi_1 U_1^*$. We see that π_1' is k-swapping idempotent:

$$w(\psi) = !^{k+1}(V)\pi!^{l_1+c_1}(V_1)!^{k+1}(W_0)\pi_1'$$

But c_1 is the lift of the discriminant γ_1 which starts at a depth strictly greater than k and ends at depth l_1. So, there exists $d_1' \geq 0$ such that $l_1 + c_1 = k + 1 + d_1'$. Thus, we finally get:

$$\begin{array}{rcll}
w(\psi) & = & !^{k+1}(V)\pi!^{l_1+c_1}(V_1)!^{k+1}(W_0)\pi_1' & \\
 & = & !^{k+1}(V)\pi!^{k+1+d_1'}(V_1)!^{k+1}(W_0)\pi_1' & \\
 & = & !^{k+1}(V)\pi!^{k+1}(!^{d_1'}(V_1)W_0)\pi_1' & \\
 & = & !^{k+1}(V)!^{k+1}(!^{d_1'}(V_1)W_0)\pi\pi_1' & \text{swapping property for } \pi \\
 & = & !^{k+1}(V!^{d_1'}(V_1)W_0)\pi\pi_1' &
\end{array}$$

Since π and π'_1 are both k-swapping idempotents, their product also is. □

Theorem 6.5.7 *Every regular wbp is legal.*

Proof Let φ be a regular wbp. We must prove that every @-cycle in φ satisfies the legality condition. In other words, let us consider any subpath of φ of the kind

$$\phi = v_1\gamma_1 u_1 \varphi_1^r @ u^r \psi u @ \varphi_2 \lambda_2 u_2^r \gamma_2^r v_2$$

where $u^r\psi u$ is a @-cycle: we must prove that the call and return paths are equal, that is

$$\psi_1 = v_1\gamma_1 u_1 \varphi_1^r @ \stackrel{?}{=} v_2\gamma_2 u_1 \varphi_2^r @ = \psi_2$$

Let k and k_i be the respective depths of @ and λ_i, for $i = 1,2$. The weights of φ_i and γ_i are respectively of the form U_i and $!^{k_i}(G_i)$ where G_i is a w-discriminant of lift c_i. Then, the weight of ϕ is

$$w(\phi) = !^{k_2}(G_2^*)!^{k_2}(p^*)U_2!^k(p)w(\psi)!^k(p^*)U_1^*!^{k_1}(p)!^{k_1}(G_1)$$

By the rendez-vous property of φ_1 and φ_2, this weight can be reduced to:

$$w(\phi) = !^{k_2}(G_2^*)U_2 w(\psi) U_1^*!^{k_1}(G_1)$$

The @-cycle property tells us that $w(\psi) = !^{k+1}(U)\pi$ for some monomial U and k-swapping idempotent π. Therefore:

$$\begin{array}{rcll} w(\phi) & = & !^{k_2}(G_2^*)U_2!^{k+1}(U)\pi U_1^*!^{k_1}(G_1) & \\ & = & !^{k_2}(G_2^*)!^{k_2+1}(U)U_2\pi U_1^*!^{k_1}(G_1) & \text{rendez-vous for } U_2 \\ & = & !^{k_2}(G_2^*!(U))U_2\pi U_1^*!^{k_1}(G_1) & \\ & = & !^{k_2}(!^{c_2}(U)G_2^*)U_2\pi U_1^*!^{k_1}(G_1) & \text{swap equation for } G_2^* \\ & = & !^{k_2+c_2}(U)!^{k_2}(G_2^*)U_2\pi U_1^*!^{k_1}(G_1) & \end{array}$$

Let $W_i = !^k(p^*)U_i^*!^{k_i}(p)!^{k_i}(G_i) = U_i^*!^{k_i}(G_i)$ be the weight of ψ_i. Then, the weight of ϕ can be equivalently expressed as

$$w(\phi) = !^{k_2+c_2}(U)W_2^*\pi W_1$$

Even if so not already know whether ψ_1 and ψ_2 are equal or not, they are two straight paths ending at the same application node. So, either $\psi_i = \phi_i n_i \phi$ where n_1 and n_2 are *distinct* premises of a binary node, or w.l.o.g. ψ_2 is a suffix of ψ_1.

In the first case, let x_i be the weight of n_i, V_i and V be the respective weights of ϕ_i and ϕ, so that $W_i = Vx_iV_i$, and

$$w(\phi) = !^{k_2+c_2}(U)V_2^*x_2^*V^*\pi Vx_1V_1$$

Since n_1 and n_2 are distinct premises of a binary node, we have $x_2^*x_1 = 0$. Moreover, $V^*\pi V$ is an idempotent. So, it is interpreted as a partial identity in the model of partial one-to-one transformations of $\mathbb{N}$. By the semantical AB* theorem we conclude that $x_2^*V^*\pi Vx_1 = 0$ and therefore $w(\phi) = 0$, contradicting the hypothesis of regularity.
Hence, we have that ψ_2 is a suffix of ψ_1. This entails that φ_2 is a prefix of φ_1. But since they are are both wbp's of type @-λ, they are forcedly equal. Thus, $\psi_i = v_i\gamma_i\lambda_1\varphi_1^r@$. Joined to the fact that ψ_2 is a suffix of ψ_1, we conclude that $\psi_1 = \psi_2$. □

6.5.3 Consistent paths are regular and vice-versa

Let us rephrase the notion of consistent path in $\mathcal{G}(M)$. According to the definition of contexts given in section 6.3, an (infinite) context can be also seen as a function from natural numbers to levels, where a level is any word of the language generated by the following grammar:

$$a ::= \ \Box \mid \circ \cdot a \mid \star \cdot a \mid \# \cdot a \mid \$ \cdot a \mid \langle a, b\rangle$$

Let $\mathcal{C}^\infty$ be the class of infinite contexts. We want to interpret the elements of the algebra $\mathcal{LS}$ as partial transformations on contexts. Namely, we want to show that there exists a set $\mathcal{H}$ of partial injective endomorphisms of $\mathcal{C}^\infty$ that is a non trivial model of $\mathcal{LS}$.

The interpretation of product and $*$ of $\mathcal{LS}$ in $\mathcal{H}$ will be the natural ones. Namely, the product of $\mathcal{LS}$ will be function composition; while for the $*$ operation, let us note that, since all the functions in $\mathcal{H}$ are injective, they are left-invertible, i.e., for any $h \in \mathcal{H}$ there is h^* such that h^*h is the identity on the domain of h. Therefore, we left to find how to interpret the constants of $\mathcal{LS}$.

As usual, let us denote by $A_n[C]$ a context of the form $\langle \ldots \langle C, a_n\rangle, \ldots a_1\rangle$.

The *basic transformations* of level n are injective partial transformations on contexts defined (for each n) in the following way:

$$\begin{aligned}
d^n(A_n[\langle C, a\rangle]) &= A_n[\langle\langle C, a\rangle, \Box\rangle] \\
t^n(A_n[\langle\langle C, a\rangle, b\rangle]) &= A_n[\langle C, \langle a, b\rangle\rangle] \\
r^n(A_n[\langle C, a\rangle]) &= A_n[\langle C, \star \cdot a\rangle] \\
s^n(A_n[\langle C, a\rangle]) &= A_n[\langle C, \circ \cdot a\rangle]
\end{aligned}$$

$$\begin{aligned} \mathtt{p}^n(A_n[\langle C, a\rangle]) &= A_n[\langle C, \# \cdot a\rangle] \\ \mathtt{q}^n(A_n[\langle C, a\rangle]) &= A_n[\langle C, \$ \cdot a\rangle] \end{aligned}$$

The family of *partial injective context transformations* $\mathcal{H}$ is then the smallest set of partial injective functions containing the previous basic transformations and closed under composition and $*$.

Remark 6.5.8 According to the previous definition, $\mathcal{H}$ must contain the nowhere defined transformation $\mathtt{0}$ and the identity transformation $\mathtt{1}$. In fact, all the basic transformations are total and for any pairs of basic transformations h_1 and h_2 of the same level n the codomains of h_1 and h_2 are disjoint. Hence, $h_1^* h_1 = \mathtt{1}$ and $h_2^* h_1 = \mathtt{0}$.

We can now directly associate an element of $\mathcal{H}$ to paths of dynamic graphs.

To each oriented edge u (w.r.t. the natural orientation) of a graph $\mathcal{G}(M)$ we associate a basic context transformation h_u in the following way:

- If u is not the premise of a node, then h_u is the identity.
- If u is the premise of a node n of depth n, then h_u is $\mathtt{d}^n, \mathtt{t}^n, \mathtt{r}^n, \mathtt{s}^n, \mathtt{p}^n$ or $\mathtt{q}^n$ when, respectively, n is a croissant, n is a square bracket, n is a fan and u is its left premise, n is a fan and u is its right premise, n is an application or a lambda node and u is its argument or variable edge, n is an application or a lambda node and u is its context or body edge.

We define then $h_{u^r} = h_u^*$ and $h_\psi = h_{u_n} \ldots h_{u_1}$, for any path $\psi = u_1 \ldots u_n$ in $\mathcal{G}(M)$, where each u_i can be either a direct or reverse oriented edge.

Definition 6.5.9 (Consistent Paths) A consistent path is a path ψ such that h_ψ is defined on some context, i.e., $h_\psi \neq \mathtt{0}$.

Proposition 6.5.10 *The family $\mathcal{H}$ is a non trivial model of $\mathcal{LS}$.*

Proof Let us define the (straightforward) interpretation of the monomials of $\mathcal{LS}$. The constants $\mathtt{0}$, $\mathtt{1}$, $\mathtt{d}$, $\mathtt{t}$, $\mathtt{r}$, $\mathtt{s}$, $\mathtt{p}$ and $\mathtt{q}$ of $\mathcal{LS}$ are interpreted respectively by $\mathtt{0}$, $\mathtt{1}$, $\mathtt{d}^0$, $\mathtt{t}^0$, $\mathtt{r}^0$, $\mathtt{s}^0$, $\mathtt{p}^0$ and $\mathtt{q}^0$; the involution $*$ is the inversion of partial transformation; the morphism $!$ (in $\mathcal{LS}$) is the morphism of $\mathcal{H}$ defined by:

$$!(h)\langle C, a\rangle = \langle h(C), a\rangle$$

for any $\mathtt{h} \in \mathcal{H}$. In particular, let us note that, if $\mathtt{b}^n$ is a basic transformation of level n, the previous definition implies that $!(\mathtt{b}^n) = \mathtt{b}^{n+1}$ and, by iteration, that:

$$\mathtt{b}^n = !^n(\mathtt{b}^0).$$

In the following, when not ambiguous, we shall drop the superscript 0 from the interpretations of the constants.
The facts that ! of $\mathcal{H}$ is a morphism and that the inversion is an antimorphism for composition are immediate. In particular, let us note that if $\mathtt{h}$ is a transformation with full domain, then $!(\mathtt{h})$ has a full domain as well. Hence, to complete the proof we left to check the axioms for the constants.
Let $C = \langle C_0, a\rangle$ be any context. Then, $\mathtt{p}^*\mathtt{p}(C) = \mathtt{p}^*\langle C_0, \sharp\cdot a\rangle = \langle C_0, a\rangle = C$. Furthermore, $\mathtt{p}^*\mathtt{q}(C) = \mathtt{p}^*\langle C_0, \natural \cdot a\rangle$. But $\langle C_0, \natural \cdot a\rangle$ is not in the codomain of $\mathtt{p}$ and thus it is not in the domain of $\mathtt{p}^*$. Therefore, $\mathtt{p}^*\mathtt{q}$ is nowhere defined, i.e., $\mathtt{p}^*\mathtt{q} = 0$.
Let $C = \langle\langle C_1, b\rangle, a\rangle$ be any context. For any $\mathtt{h} \in \mathcal{H}$, we have that $\big(!(\mathtt{h})\mathtt{t}\big)(C) = !(\mathtt{h})\langle C_1, \langle a, b\rangle\rangle = \langle \mathtt{h}(C_1), \langle a, b\rangle\rangle$ and that $\big(\mathtt{t}!^2(\mathtt{h})\big)(C) = \mathtt{t}\langle\langle \mathtt{h}(C_1), b\rangle, a\rangle = \langle \mathtt{h}(C_1), \langle a, b\rangle\rangle$. So, $!(\mathtt{h})\mathtt{t} = \mathtt{t}!^2(\mathtt{h})$.
The computations for the other axioms are similar and we left them as an exercise. □

Theorem 6.5.11 *A path φ in $\mathcal{G}(M)$ is consistent iff it is regular.*

Proof The interpretation of $w(\psi)$ in $\mathcal{H}$ is clearly $\mathtt{h}_\psi$. By definition, ψ is consistent if and only if $\mathtt{h}_\psi \neq 0$. Then the only if part of the theorem follows from the fact that $\mathcal{H}$ is a model of $\mathcal{LS}$. The if part is a consequence of Corollary 6.4.3, that is, a consequence of the the semantical AB* theorem. □

Remark 6.5.12 The model $\mathcal{H}$ is not initial in the category of dynamic algebras. For instance $\mathtt{d}^*\,\mathtt{t}$ is not provably equal to 0 in $\mathcal{LS}$. But it is equal to 0 in $\mathcal{H}$, since the transformation $\mathtt{d}^*\mathtt{t}$ is nowhere defined.

6.6 Virtual interactions

In Section 6.2 we proved that Lévy's families of redexes for a λ-term M can be described as suitable *paths* (virtual redexes) in (the syntax tree of) M. As a consequence, since Lamping's graph reduction technique

is optimal, we have a bijective correspondence between Lamping's β-interactions and virtual redexes. A similar result can be obtained for the other crucial annihilation operation of sharing graphs: the interaction between fans of a same level (and, more generally, for all kinds of annihilations rules). In particular, we shall prove that fan-annihilations are in one-to-one correspondence with paths starting and terminating *at the same fan* in M and traversing, in order, a virtual redex ψ, an @-cycle and ψ^r (i.e., the path ψ but in the reverse direction).

Let us consider for instance the "abstract" reduction of the term $(2\ \Delta)$ depicted in Figure 6.16. In this reduction, we have applied two fan-annihilation rules. Following the analogy with β-reduction and virtual redexes, our problem is to describe these interactions as suitable paths in the initial term of the reduction. The general idea is very simple: consider the annihilation rule and, going backward along the reduction, follow the paths traversed by the two interacting fans. Applying this intuition to our example, we discover that the two annihilation rules in Figure 6.16(7) correspond to the paths in the initial term illustrated in Figure 6.17.

These paths have a very precise and similar structure: they both start and terminate at the same fan, and can be uniquely decomposed as a virtual redex ψ, followed by an @-cycle (see Definition 6.2.21 in Section 6.2) and by ψ-reverted. This decomposition is general, as we shall see. In this way, we can recast (and explain) all dynamical aspects of Lamping's algorithm in terms of statics.

6.6.1 Residuals and ancestors of consistent paths

The first step is to formalize the notion of residual (and ancestor) of a consistent path in a sharing graph.

Let $[M] \to^* G \xrightarrow{u} G'$ and ρ be a path in G which starts and terminates at two principal ports †. The notion of residual path is not defined when u is an annihilation rule and u involves the endpoints of ρ.

By definition of sharing rules, u is "local", namely it involves exactly two nodes n and n' in G and the edges starting at these nodes. Therefore, let G_{flat} be the subgraph of G where n, n' and the edges starting at n and n' are missing. Then G_{flat} is also a subgraph of G'. If ρ is internal to G_{flat} of G then the *residual* of ρ is the corresponding path in G_{flat}

† This constraint guarantees the uniqueness of the ancestor. Indeed, assume ρ starting at the auxiliary port of a node m, m is involved in u and u is not the first edge of ρ. Then the residuals of ρ and $u\rho$ should be the same.

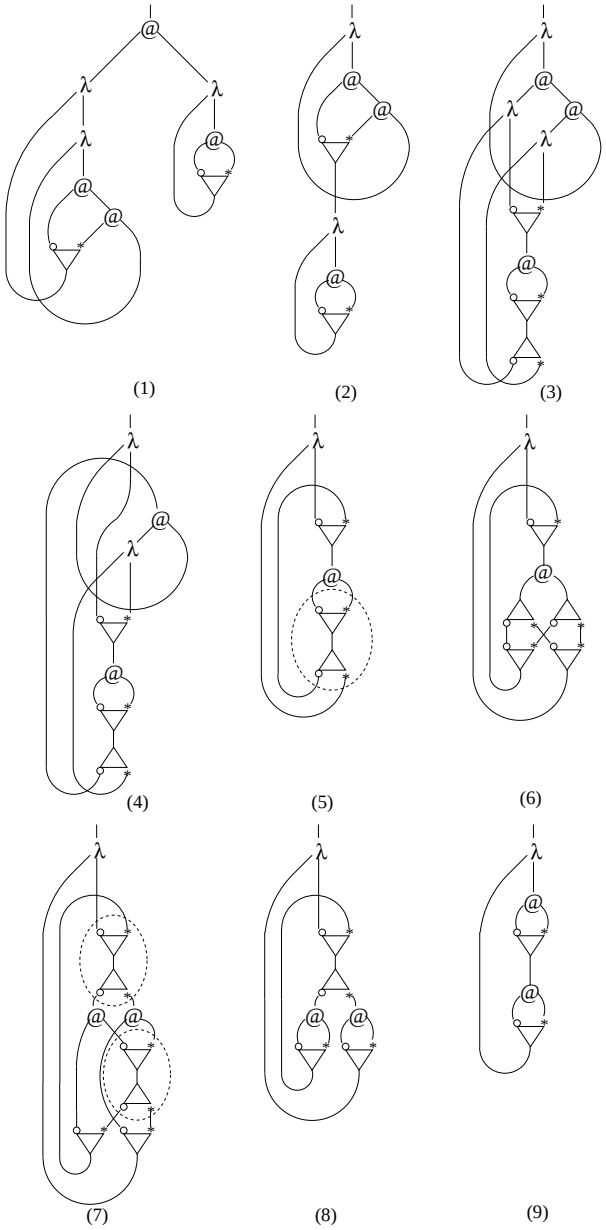


Fig. 6.16. Abstract reduction of $(2\ \Delta)$

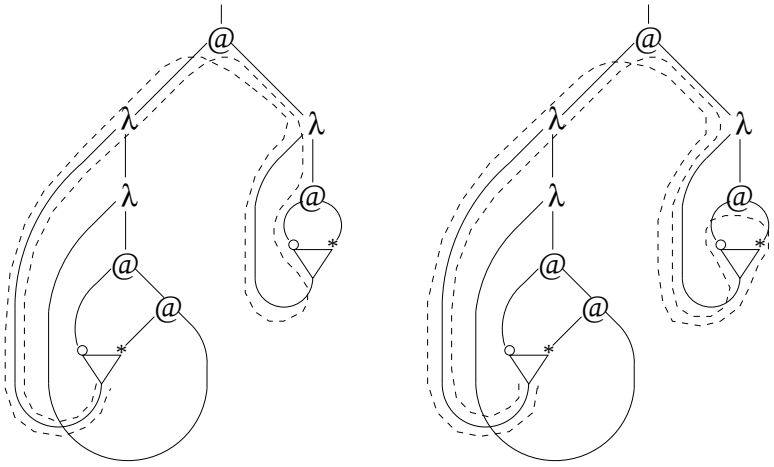


Fig. 6.17. Virtual fan-annihilations in $(2\ \Delta)$

of G'. Otherwise $\rho = \rho_1 e_1 m_1 u p_1 e'_1 \cdots e_k m_k u p_k e'_k \rho_{k+1}$ such that ρ_i are internal to the subgraph G_{flat} of G, $\rho_1 e_1$ or $e'_k \rho_{k+1}$ may be missing and $\{m_i, p_i\} = \{n, n'\}$. There are two cases:

(u *is a commutation rule*) Let us define the cases when $\rho_1 e_1$ or $\rho_1 e_1 m_1 u$ are missing: the other cases may be defined in a similar way. In this case the residual of ρ is $m'_1 c'_1 \rho'_2 \cdots c_k p'_k \nu_k m'_k c'_k \rho'_{k+1}$, where ρ'_i are the residuals of ρ_i, $c_i p'_i \nu_i m'_i c'_i$ are the unique paths traversing the part of G' which is not in G_{flat} such that they connect the ports which correspond to the initial port of e_i and the final port of e'_i. The node m'_1 is consecutive to the initial node of ρ'_2 through the port corresponding to the final port of e'_1.

(u *is an annihilation rule*) Remark that $\rho_1 e_1$ and $e'_k \rho_{k+1}$ cannot be missing in this case. The residual of ρ is $\rho'_1 c_1 \cdots c_k \rho'_{k+1}$, where ρ'_i are the residuals of ρ_i and c_i are the edges connecting the ports which correspond to the initial port of e_i and the final port of e'_i.

There is an obvious consequence of the above definition:

Proposition 6.6.1 *The residual of a path (if any) is unique.*

The uniqueness of residuals allows to define the "inverse" notion, called ancestor.

Definition 6.6.2 Let σ be the residual of ρ. Then ρ is called the *ancestor* of σ.

Obviously, since Lamping's graph rewriting rules preserve the consistency of paths, the residual of a consistent path is still consistent. Similarly for the ancestor.

6.6.2 Fan annihilations and cycles

Lemma 6.6.3 *Let φ be a path starting and terminating at the principal ports of two abstractions λ and λ'. Let ξ and ξ' be two discriminants of λ and λ', respectively, such that $\xi\varphi$ and $\varphi(\xi')^r$ are both legal. The path $\xi\varphi(\xi')^r$ is illegal if and only if:*

(i) $\lambda = \lambda'$, $\xi = \zeta u \chi$, $\xi' = \zeta v \chi'$ *and* $u \neq v$;

(ii) $\varphi = \lambda\psi@\phi@\psi^r\lambda$, *where ψ is a virtual redex and ϕ is an @-cycle.*

Proof The if direction is immediate by definition of legality. The only-if direction is proved by contradiction. Let $\xi\varphi(\xi')^r$ be illegal and $\xi\varphi(\xi')^r \neq \chi u \zeta\lambda\psi@\phi@\psi^r\lambda\zeta^r v(\chi')^r$, with $u \neq v$, ψ be a virtual redex and ϕ be a @-cycle. Since $\xi\varphi(\xi')^r$ is illegal, there exists a subpath $\vartheta = \sigma\lambda_1\rho@'\phi'@'(\rho')^r\lambda_2(\sigma')^r$ of $\xi\varphi(\xi')^r$ such that $\rho \neq \rho'$ or $\sigma \neq \sigma'$. Now, remark that ϑ must be a subpath of $\xi\varphi$ or of $\varphi(\xi')^r$, which implies that one between $\xi\varphi$ and $\varphi(\xi')^r$ is illegal. But this is impossible, since they are both legal, by hypothesis. □

Theorem 6.6.4 *Let $[M] \rightarrow^* G$ and u be an edge in G connecting the principal ports of two fans. The interaction u annihilates the two fans if and only if the ancestor of u is a path $\xi\lambda\psi@\phi@\psi^r\lambda\xi^r$, where ξ is a discriminant, ψ is a virtual redex and ϕ is a @-cycle.*

Proof

(*if direction*) Let $\rho = \xi\lambda\psi@\phi@\psi^r\lambda\xi^r$ be the ancestor of u such that ξ is a discriminant, ψ is a virtual redex and ϕ is a @-cycle. Then *every path* traversing the $*$-port of an ending fan of ρ and the $\circ$-port of the other is illegal (hence inconsistent, by Theorem 6.5.4 and Theorem 6.5.11). Therefore the interaction u cannot commute the two fans, otherwise the ancestor of a consistent path could be inconsistent, invalidating the context semantics.

(*only-if direction*) Assume that $\mathfrak{u}$ annihilates the two fans. This means that *every path* traversing the $*$-port of a fan and the $\circ$-port of the other is inconsistent, since sharing rules preserve consistency (by the context semantics). Now, take the ancestor ρ of $\mathfrak{u}$ in $[M]$. Notice that $\rho = \xi\lambda\varphi\lambda'(\xi')^r$. Moreover, ρ is consistent because $\mathfrak{u}$ is consistent. Let e be the edge of the $*$-port of one of the ending fans of ρ and e' be the edge of the $\circ$-port of the other. It is not difficult to show that $e\rho$ and $\rho e'$ are consistent. However, $e\rho e'$ is inconsistent, since its residual in G will traverse the $*$-edge of a fan of $\mathfrak{u}$ and the $\circ$-edge of the other. Then, by Lemma 6.6.3 (only-if direction), $\xi = \xi'$ and $\varphi = \lambda\psi@\phi@\psi^r\lambda$, where ψ is a virtual redex and ϕ is a @-cycle.

□

Corollary 6.6.5 *If two fans match (are paired), they are residual of a same fan in the initial graph.*

All previous results could be easily generalized to the other control operators (do it as an exercise).

Let us remind that the converse of Corollary 6.6.5 is not true in general. In fact, there are cases in which two residuals of a same control node meet with different indexes, and consequently, they do not annihilate. This is indeed the reason because of which a static labeling of fans was insufficient and we had to introduce indexes and brackets (see the example in Figure 3.2 at the beginning of Chapter 3). Nevertheless, the previous corollary shows that all the control nodes in an initial graph are indeed distinct objects, for two control operators with distinct ancestors in the initial graph will never match. For instance, we could explicitly distinguish all the control nodes in the initial graph decorating them with a distinct name. Preserving the usual operational behavior of control nodes, we could also assume that names are invariant along reduction (i.e., a control node bear a given name iff it is the residual of a node with such a name). In the case of facing control operators we could then check if they have the same index—proceeding thus with an annihilation—only when they have the same name; in the other cases, we could instead directly proceed with a swap, for by Corollary 6.6.5 the nodes have definitely different indexes. Such a property will be exploited by the introduction of *safe operators* (Chapter 9) to prove soundness of some additional simplification rules of control nodes. In particular, we will define *safe* an operator whose residuals may erase only by anni-

hilation rules involving two of them. Therefore, according to such a definition, Corollary 6.6.5 states that all the control nodes in the initial graph are safe.

The theoretical interest of Theorem 6.6.4 is that it allows to translate a dynamical concept (an interaction) into a static one (a path). This correspondence between paths and annihilations explains in a very simple and intuitive way *why* fan annihilates (due to the legal, looping nature of the path), *which* fans will annihilate (residuals of a same initial fan along the path), and also, by a simple inspection of the path, *where* and *when* the actual annihilation will take place.

7
Read-back

In this chapter we will pursue a more syntactical study of the properties of sharing graphs and we will present a way to group control nodes allowing to solve and simplify the problem of read-back.

The set of rules for control nodes that we already met will be extended by other rules preserving the correspondence between sharing graphs and λ-terms. The extended set of *propagation rules* obtained in this way will not be anymore optimal, but will allow a clean study of Lamping's technique in terms of usual rewriting system properties. Moreover, given the previous extended set of propagation rules, say π, we will see that Lamping's optimal algorithm corresponds exactly to the subset of rules induced by a *lazy* application (w.r.t. duplication of β-redex edges) of the rules in π.

The analysis moves from the idea that the propagation rules for control nodes are a distributed and step-by-step implementation of β-rule. According to this, a natural requirement might thus be that, for any β-rule $M \to N$, there exist a corresponding sharing graph reduction $[M] \twoheadrightarrow [N]$ rewriting the translation of M into the translation of N. We stress that this is much stronger than what we have got proving soundness of Lamping's algorithm. There we just showed that for any sharing reduction $[M] \twoheadrightarrow G$ the graph G is some compact representation of a reduct N of M (i.e., $M \twoheadrightarrow N$ in λ-calculus). The previous requirement asks instead to reach the translation of N at the end of some reduction of G (i.e., $G \twoheadrightarrow [N]$).

The existence of such a reduction would give by definition an internal implementation of the read-back algorithm. At the same time it must be clear that there is no chance to get this without extending the rewriting rules. Namely, the rules implementing read-back will not necessarily be optimal preserving.

A relevant case in which the read-back property holds without any addition to the usual optimal rules is the term $(\delta\ \delta)$ (see Figure 3.2 in Chapter 3). As we already know, this λ-term always rewrites to itself. Moreover, being $G = [(\delta\ \delta)]$, it is not difficult to see that after the contraction of the unique β-redex in G we get a shared representation G' of $(\delta\ \delta)$ that can at its turn be reduced to G by a sequence of bracket and fan rules (i.e., $G \to_\beta G' \twoheadrightarrow G$). In this case, things work well since, unfolding the shared instances of δ, we do not have to duplicate any virtual redex. As a matter of fact, the optimal rules will not allow to achieve the previous property in the general case. For instance, let us take the redex $R = (\lambda x.\, M\ N)$ such that N contains at least a redex, and x occurs several times in N. By definition of optimality, if $(\lambda x.\, M\ N) \xrightarrow{R} T$, no optimal sharing graph implementation can admit a reduction $[(\lambda x.\, M\ N)] \twoheadrightarrow [T]$.

Later, we will indeed see that, without any extension to the basic set of rules given in Chapter 3, the previous problem is unsolvable independently of sharing. Namely, in absence of sharable redexes, the reduction may end with an unshared representation of a λ-term T that differ from $[T]$ because of some spurious control nodes accumulated along the reduction (for the full details and some examples see Chapter 9).

Summarizing, let us proceed forgetting optimality for a while. We aim at finding all the propagation (π) rules for control nodes that are sound w.r.t. the interpretation of sharing graphs as λ-terms. Given such a set of rules we will study its main properties (confluence and strong normalization). Finally, we will notice that Lamping's algorithm is just a particular lazy reduction strategy for π-rules.

As a main result we will get that, given a λ-term M, for any sharing reduction $[M] \twoheadrightarrow G$, the extended set π of control node rules allows to reduce G to the sharing graph $[N]$, where N is the λ-term matching with G. In particular, assuming that $[M] \twoheadrightarrow G$ is an optimal reduction, the unrestricted application of the rules π gives an algorithm for the read-back of the λ-term corresponding to G.

Let us also remark that, since the proof of correctness of the read-back of G will rest on the fact that $G \twoheadrightarrow [N]$ for some λ-term N such that $M \twoheadrightarrow N$, this will give another correctness proof of Lamping's algorithm.

The previous approach generalizes to more general graph rewriting systems in which there is a rule similar to β of λ-calculus, say β-like. Using the natural generalization of the rules π to such systems, the global β-like rule can be implemented via a local and distributed graph

rewriting system for which, when it makes sense, optimal reduction can be defined as a particular reduction strategy for $\pi + \beta$. For a treatment of this general result we refer the interested reader to [Gue97].

7.1 Static and dynamic sharing

To present the results described in the introduction to the chapter we will resort to a small change in the presentation of sharing graphs. The following discussion will point out that in sharing graphs there are coherent sequences of control nodes which may be usefully grouped into compound operators. Similar arguments will be at the base of the safe operators that we will study in Chapter 9.

7.1.1 Grouping sequences of brackets

It is immediate to see that the following configuration:

can be interpreted as a compound operator with a null effect on the nodes it acts on. A similar argument applies to sequences of brackets with the more general shape in Figure 7.1, where $q \geq -1$.

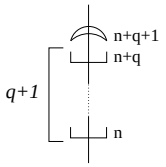


Fig. 7.1. A bracket-croissant sequence

In this case, the result is a compound node whose task is to lift by the *offset* q the nodes it acts on. In fact, let such a bracket-croissant sequence meet a generic node whose index i is greater than n. There is a rewriting sequence by which all the brackets cross the node (see Figure 7.2): each of the $q + 1$ square brackets increasing by 1 the index of the node; the croissant decreasing by 1 such an index.

By the way, even more complicated configurations can be treated as a unique compound node. Besides, their grouping turns to be useful only if such sequences correspond to patterns arising in sharing graphs

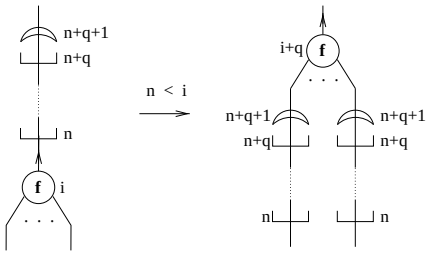


Fig. 7.2. A sequence of brackets crossing a generic node.

representing λ-terms. Let us illustrate by an example the only relevant case: the trees of brackets and fans corresponding to the occurrences of a variable in a term.

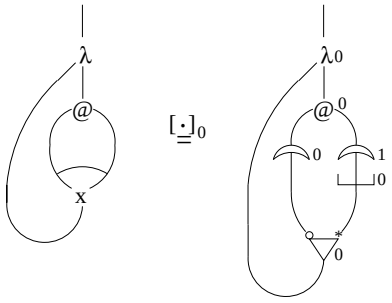


Fig. 7.3. $\delta = \lambda x.xx$.

In Figure 7.3 we have drawn the syntax tree of $\delta = \lambda x.(xx)$ (on the left) and the corresponding sharing graph (on the right). The variable x occurs twice in the body of δ. Therefore, back-connecting the occurrences of this variable to the corresponding binder we would get two binding edges incoming to the node of λx. In order, to keep the fact that any abstraction has a unique binding port, let us add in the syntax tree of δ a node x contracting the two occurrences of the variable x into a unique wire.

In terms of sharing graphs, the latter node for a variable corresponds to a fan with two sequences of brackets at its input ports. Also such fan-brackets tree may be grouped in a compound node. In fact, let the fan interact with a node. After the duplication of the node, each bracket-croissant sequence connected to an input of the fan may cross the new instance of the node, lifting it by the corresponding sequence offset.

More generally, for any variable we have a fan-bracket tree as the one in Figure 7.4, any branch of which, a *discriminant* of the variable x, is a connection from an occurrence of x to its binder.

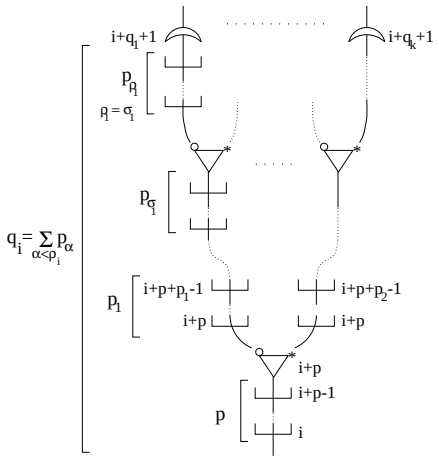


Fig. 7.4. Fan-bracket tree.

A part for fans, crossing a discriminant we meet a sequence of brackets whose offset is equal to the difference between the index at which the corresponding occurrence of x is translated (i.e., the index of the croissant) and the index of the λ node binding x decreased by 1. For instance, in Figure 7.5 we have explicitly depicted the variable x in δ.

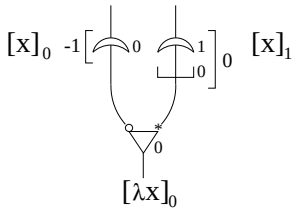


Fig. 7.5. The fan-bracket tree representing x in δ.

The tree in Figure 7.4 globally behaves as a k-ary duplicator followed by k reindexing operators lifting the i-th instance of the duplicated node by the offset q_i. To make such a behavior more explicit, let us consider the permutation rule in Figure 7.6. This rule is not justified by the

context semantics. In spite of this, the permutation is sound if applied to the initial configuration (or to so-called safe operators, see Chapter 9).

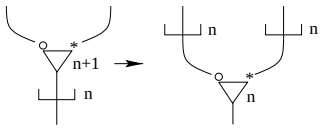


Fig. 7.6. Bracket-mux permutation—sound at initialization.

Exercise 7.1.1 Prove that the rule in Figure 7.6 is not justified by the context semantics but it is sound for a sharing graph which is the translation of a λ-term: the permutation preserves the three properties used proving correctness of Lamping's algorithm.

Iterating the transformation in Figure 7.6 any fan-bracket tree of a variable can be converted into a tree in which the brackets are pushed towards the leaves and all the fans have the same index. The compound node corresponding to such a kind of tree is the v node drawn in Figure 7.7, that is, a node with a unique output wire to be connected to the binder of the variable and an input wire with an associated offset for each occurrence of the variable.

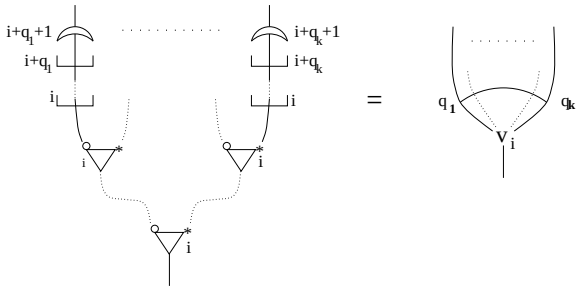


Fig. 7.7. v node

The output edge leaves a v node from its *principal port*, the input edges enter into a v node through its *auxiliary ports*. The auxiliary ports of a v node are not named, the only label they have is an *offset* greater or equal than -1. Thus, two ports of a v node with the same offset are indistinguishable.

7.1.2 Indexed λ-trees

Using ν nodes, we associate to each λ-term a structure tightly related to the term abstract syntax tree. Let us call such a structure an *indexed* λ-*tree*, or a λ-tree for short. There is a certain abuse of notation in defining the previous structure a tree, as it contains the loops created by the back-connections from the variable occurrences to their binders. Anyhow, just removing them, we get a tree. (For an exposition in which there is a more explicit distinction between the tree of the λ-term and its binding relation see [Gue95, Gue96].)

Given a λ-term, we could get a λ-tree applying first the translation in section 3.3 and then the permutation and grouping procedures previously described. However, the indexes of a λ-tree can be directly determined decorating the edges of the tree by an indexed term, in accord with the rules in Figure 7.8.

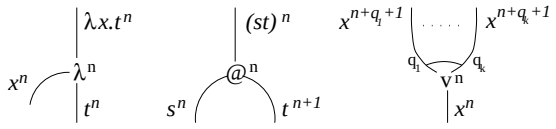


Fig. 7.8. Indexing rules.

The rules in Figure 7.8 force the uniqueness of the decoration of a λ-term syntax tree (see Figure 7.9 for the decoration of $(\delta\,\delta)$). In particular, the index n of an occurrence of a variable is greater or equal than the index l of the λ node binding it. Hence, the offset of the corresponding edge entering into a ν node is $q = n - l - 1 \geq -1$. We also stress that the previous decoration is not part of the graph: it is just useful to determine the offsets of the wires entering into a ν node.

The usual λ-calculus β-rule naturally induces a corresponding λ-tree reduction.

Definition 7.1.2 Let T be the λ-tree of the λ-term t. If T' is the λ-tree of t' and $t \to t'$, then $T \to_{\beta_\lambda} T'$.

7.1.3 Beta rule

The β-rule is implemented in the usual way (see Figure 7.10): the @-λ redex disappears substituted by direct connections from the context edge of the @ node to the body edge of the λ node, and from the binding edge

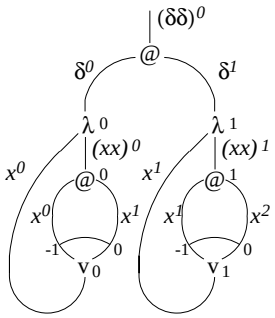


Fig. 7.9. Decoration of (δ δ).

of the λ node to the argument edge of the @ node. The main difference is that a β-reduction changes the status of the ν node connected to the binding port of the redex. In fact, such a node is no more an operator contracting the occurrences of a variable; it becomes instead a sharing node (the triangle in Figure 7.10) contracting a set of access pointers to a same shared part of the graph.

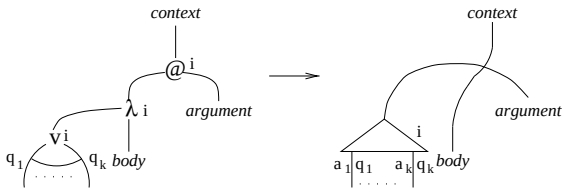


Fig. 7.10. β-rule

Hence, β-rule becomes a sort of task activator transforming a *static* object—a ν node—into a *dynamic* operator whose task is to duplicate the argument of the redex. In more details: ν nodes are introduced by the initial translation of the term; their principal ports point to the binding port of their corresponding λ node. On the converse: dynamical operators are introduced by β-rules; they interact with the nodes they point to duplicating them; during their propagation they originate unsharing operators, that is, fan-out's.

7.1.4 Multiplexer

The triangle introduced by the β-rule is a sort of *multiplexer*, or *mux* for short. A mux node is similar to a v node—it has a *principal port* for the outgoing edge and a set of *auxiliary ports* for the incoming edges—but, differing from v node, the auxiliary ports of a mux are named and distinguishable, see Figure 7.11. Namely, the k auxiliary ports of a mux are labeled by names $a_1, \ldots, a_k$ chosen among a denumerable set of names A with the proviso that $a_i = a_j$ implies $i = j$.

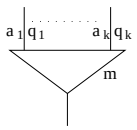


Fig. 7.11. Multiplexer (mux) node

As for fans, we have positive and negative muxes. The interpretation is the same, a positive mux actually denotes a multiplexing operator, a negative mux is rather a sort of demultiplexing operator.

The index m of a mux is its *threshold*. The reasons of such a name will become clear after the analysis of the mux rewriting rules. In fact, we shall see that a mux duplicates an i-indexed node only if i is above its threshold (i.e., $m < i$).

Remark 7.1.3 We restrict our analysis to λI-calculus. This implies that any v node, and thus any mux too, has at least an auxiliary port.

In the following we will need to distinguish among unary and k-ary muxes. In particular, we will refer to the first ones as positive/negative *lifts*. Although a lift is not a duplicator but just an operator for index bookkeeping, it cannot be simply removed from sharing graphs. In fact, after a β-redex involving a linear variable, the corresponding lift has the task to increase by its offset all the indexes in the argument of the redex.

The interpretation of a mux in terms of fans and brackets is exactly the same of a v node. Namely, a tree of fans followed by suitable sequences of brackets. This suggests a natural way to choose the names for its auxiliary ports. If A is the set of the strings over $\{*, \circ\}$, let us take for a_i the string corresponding to the discriminant path connecting the root of the tree to its i-th leaf. Such a solution implies that in the unary case we have only one possible choice: $a = \epsilon$ (the empty string). However, for the successive proofs, we prefer to assume a more general assignment for

the names a_i that do not force an empty value in the unary case. As a matter of fact, from the implementation point of view, we could simply get rid of mux auxiliary port names, assuming that their auxiliary ports are numbered according to some ordering.

7.2 The propagation rules

Hitherto, the aim was at developing implementations techniques for optimal reductions. Such a requirement was fulfilled enforcing a lazy behavior of control nodes. Dropping such a constraint, we can see that beyond the optimal rules, there are other rewriting rules (again, see also Chapter 9) that can be added to the system with no impact on soundness, although some of them causing the loss of optimality.

Thus, let us change the focus. We go on without paying attention to optimality, but rather concentrating on the dynamical interpretation of muxes as duplicators. In this way, we achieve a set of rewriting rules for muxes, say π-*rules*, composed of all the local graph rewritings sound w.r.t. the intended interpretation of sharing graphs in λ-calculus.

Given such an extended set of rewriting rules, the natural question is: what can we say about the confluence and normalization of them? The answer is: restricting our attention to the sharing graphs obtainable starting from a λ-tree (further we implicitly assume this as definition of sharing graph), π-rules are confluent and strong normalizing. Furthermore, the (unique) π normal-form of a sharing graph G is the λ-tree corresponding to G, that is, its read-back.

7.2.1 Mux interactions

The first π-rules we consider deal with the case of a facing pair of muxes. Such rules are the direct reformulation of the usual ones for fans and brackets, see Figure 7.12.

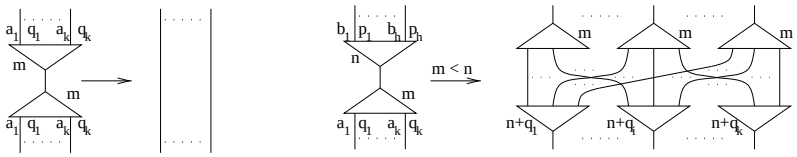
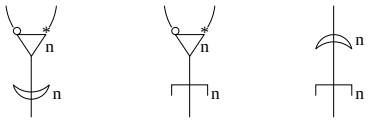


Fig. 7.12. Mux interactions: annihilation (left) and swap (right).

To apply the annihilation rule the pair has to be composed of *com-*

plementary muxes. It does not suffice to ask that the thresholds of the muxes coincide, their set of ports must have matching name-offset pairs (i.e., for any port of the mux labeled with the name-offset pair (a_i, q_i), there has to be a corresponding port of the negative mux labeled by (a_i, q_i), and vice versa). After the annihilation, the pair of muxes is replaced by a set of direct connections between the edges formerly connected to matching ports. Let us note that the rule is not ambiguous because of the uniqueness of auxiliary port names.

A mux pair with the same threshold but with a mismatch between the auxiliary ports is a deadlock, that is, a configuration irremediably stuck, for it cannot be removed by applying any of the rewriting rules. Such kind of situations were already present in the formulation using brackets and fans:



In particular, the deadlock caused by a mismatch between cardinalities corresponds to a deadlocked configuration composed of a fan with index n facing with a croissant/bracket having the same index n. The deadlock caused by a mismatch between the labels of the auxiliary ports corresponds instead to a bracket and a croissant meeting with the same index.

7.2.2 Mux propagation rules

As a consequence of the interpretation of a mux as an unrestricted duplicator, there is a rule for any case in which the principal port of a mux is connected to a logical node ($\lambda, @, \nu$ nodes).

In the case in which the logical node is a λ or an @, the analysis is immediate: such nodes cannot be the border of the duplicating subgraph, it is then sound to allow muxes to duplicate them in any case. The corresponding set of mux propagation rules is drawn in Figure 7.13. The side condition $m < i$ restricts the applicability of the rules to the case in which the index of the λ-@ node is above the threshold of the mux. Most of the cases excluded by the side condition will be easily proved unreachable (see Exercise 7.2.4). However, such simple reasonings will not suffice to prove that a mux with threshold $m = i$ cannot reach the argument port of an @ node with index i. Let us note that a situation

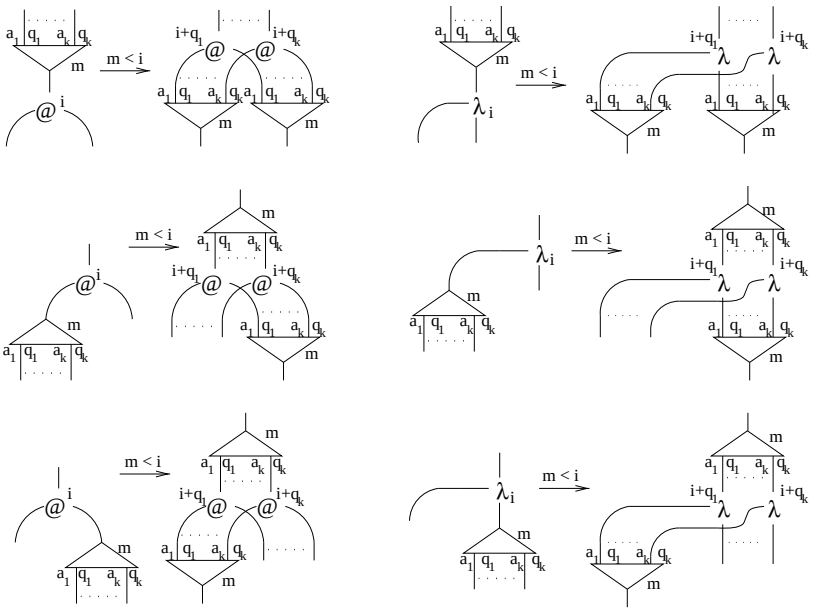


Fig. 7.13. Propagation rules

like this would lead to a deadlocked configuration after the execution of a β-rule involving the @ node pointed by the mux. To exclude the presence of such deadlocks, we will need to develop the algebraic semantics of section 7.7

The analysis of the interactions between a mux and a v node deserves instead more attention. When the side condition $\mathfrak{m} < \mathfrak{i}$ holds, we can proceed as for the other logical nodes, obtaining the propagation rules drawn in Figure 7.14.

The problem with the left-hand sides of the rules in Figure 7.14 is that it is no more true that the side condition $\mathfrak{m} < \mathfrak{i}$ always hold in sharing graphs. In fact, at the end of its propagation, a mux might reach the v node of a variable whose binder is external to the duplicating subgraph, that in terms of indexes, means with $\mathfrak{m} \geq \mathfrak{i}$. This has no counterpart for @ and λ-nodes and deserves indeed a rule ad-hoc. In fact, since this case corresponds to a mux reaching the border of the subterm it had to duplicate, the operator should stop duplicating there. In some sense, the mux should be erased, for it has completed its task.

To better understand what to do in these cases, and to see the π-rules at work, let us proceed with an example.

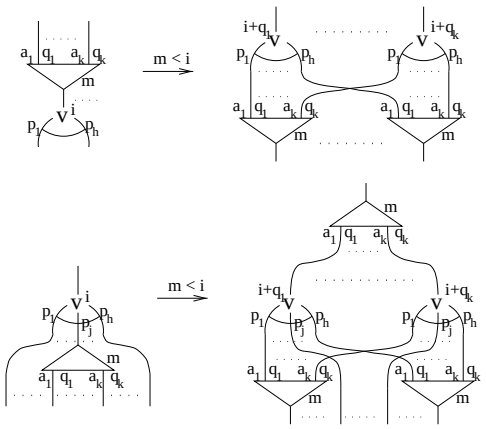


Fig. 7.14. Mux-ν propagation rules.

7.2.3 The absorption rule

Let us take the term $\lambda z.\,(\lambda w.\,(w\,w)\;\lambda x.\,\lambda y.\,(y\;(x\,z)))$. Its λ-tree and the initial steps of its reduction are drawn in Figure 7.15. At the end of this reduction the mux named α in Figure 7.15 is at the auxiliary port of a ν node. We fired just one β-rule, all the muxes are copies of the one introduced by this rule, their task is thus the duplication of the term $T = \lambda x.\,\lambda y.\,(y\;(x\,z))$. The ν node pointed by α corresponds to the variable y, which is bound in T. Then, not only the occurrence of y has to be duplicated, but its binder too. (Note that this duplication could be equivalently performed by any of the muxes $\bar{\alpha}$. In the optimal algorithm, by the top one.) Since y is bound in T and the mux was created outside it, that is, at a level lower than T, the threshold of α is smaller then the index of the ν node; the side condition of the ν-propagation rule holds; the corresponding reduction is the one in Figure 7.16.

After the annihilation of the two pairs of muxes $\alpha, \bar{\alpha}$ created by the latter reduction, we get the sharing graph on the left in Figure 7.17, in which only the mux γ is left. Also γ points to a ν node, but in this case the node represents a variable (z) whose binder is external to T. In other words, γ has reached the border of its scope and completing its task; in terms of indexes the threshold of the mux is greater or equal than the index of the ν node. Therefore, γ can be incorporated into the ν node of z, and the edges entering into γ can be directly connected to the ν node of z, adding to their offsets the offset of the port previously

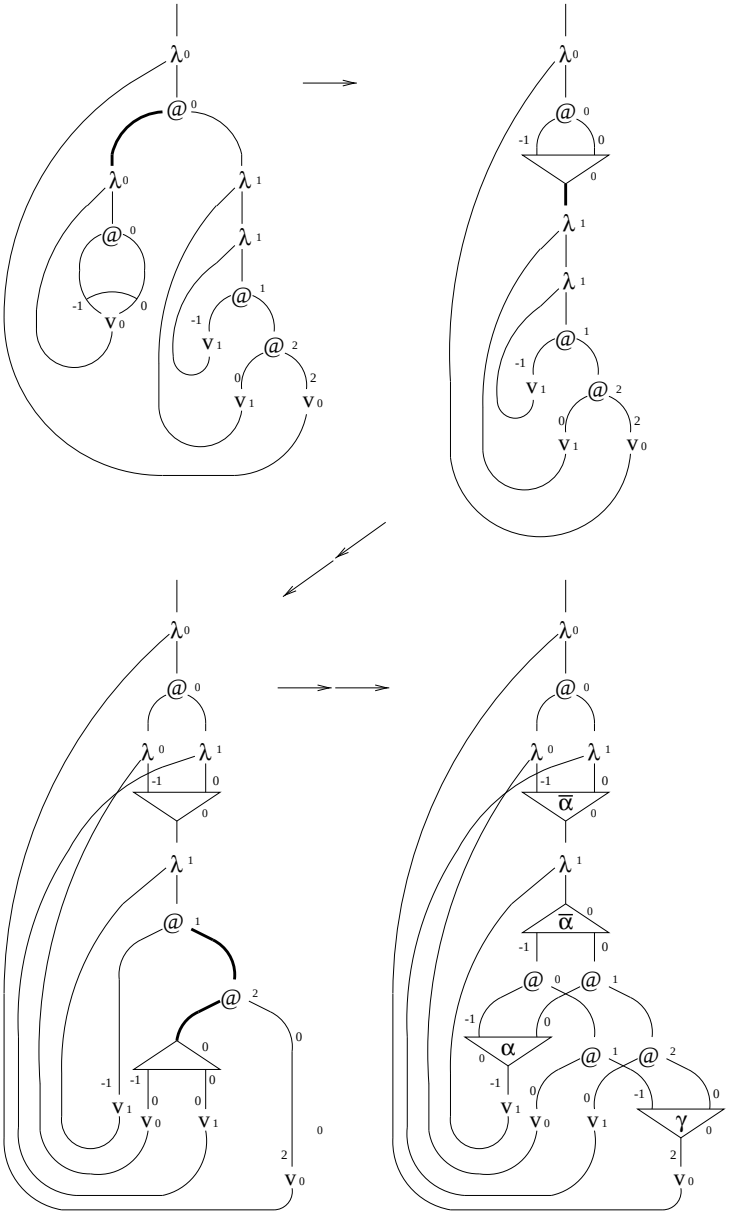


Fig. 7.15. Reducing $\lambda z.(\lambda w.(ww)\ \lambda x.\lambda y.(y\ (x\ z)))$

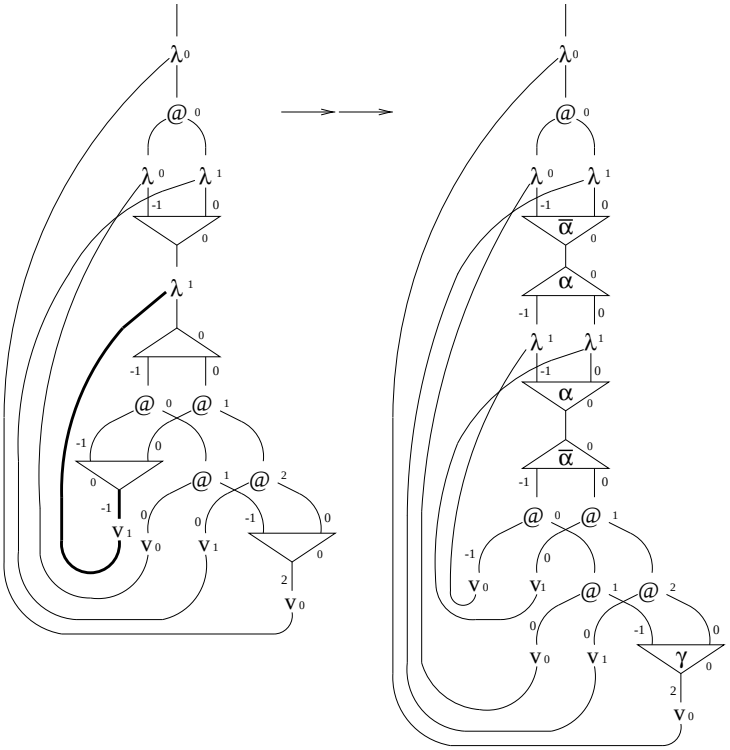


Fig. 7.16. An example of backward v-mux propagation

pointed by γ. The result of such a reduction is the sharing graph on the right in Figure 7.17.

Summarizing, when a mux with threshold m reaches an auxiliary port of an n-indexed v node, we can have:

- $m < n$, that is, the occurrence of the variable and its binder are both in the scope of the mux. The mux can propagate duplicating the v node as described by the bottom picture in Figure 7.14.
- $n \leq m < n+p+1$, where p is the offset of the port to which the mux is connected. In this case, the λ node binding the variable is not in the scope of the mux. The mux can complete its task just duplicating the occurrence of the variable it points to. The corresponding rule is drawn in Figure 7.18. In the reduction the mux disappear *absorbed* by the v node; its auxiliary ports replace the port of the v node to which the mux was connected, and their offsets are increased by p.

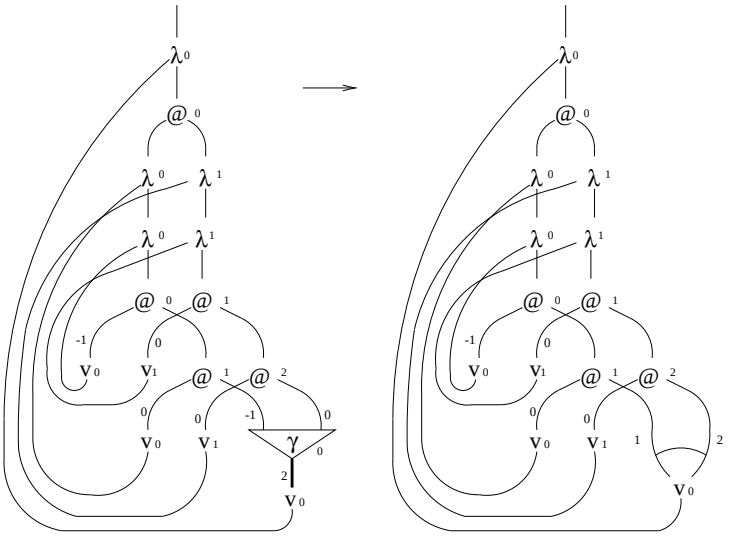


Fig. 7.17. An example of mux absorption rule

Remark 7.2.1 In the case of the mux absorption, the side condition $n \leq m < n + p + 1$ implies $p \geq 0$. Then, the offsets of the new ports of the v node are definitely greater or equal than -1.

7.2.4 Redexes

The set of the π-rules composes of the mux interaction rules in Figure 7.12, of the mux propagation rules in Figure 7.13 and Figure 7.14, and of the absorption rule in Figure 7.18. In π the only node whose interaction port is always fixed remains the mux: a mux interact with another link only through its principal port. Moreover, each mux whose principal port is not connected to an auxiliary port of another mux is a redex, provided that the proper side condition holds.

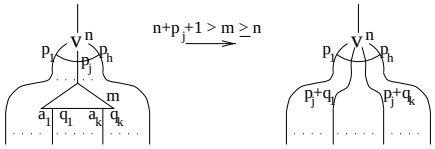


Fig. 7.18. Mux-v absorption (bottom).

Definition 7.2.2 (π-redex) A π-*redex* is a pair of nodes (α_1, α_2) in which:

(i) α_1 is a mux connected to any port of α_2 but an auxiliary one when α_2 is a mux too;

(ii) if m is the threshold of α_1 and n the threshold/index of α_2, then the following *side condition* holds:

  (a) if α_2 is a mux and $m = n$, then the sets of pairs labeling the auxiliary ports of α_1 and α_2 coincide;

  (b) if α_2 is a $\vee$ node and α_1 points to an auxiliary port of α_2 with offset q, then $m < n + q + 1$;

  (c) otherwise, $m < n$.

Muxes and $\vee$ nodes have analog interpretations in terms of brackets and fans. In spite of this, we have more reduction rules for $\vee$ nodes than for muxes. In particular, there are rules for the cases in which a mux reaches the auxiliary port of a $\vee$ node. The corresponding rules for muxes would not be sound. For instance, the absorption rule would fail. In fact, let μ be obtained by merging a mux α with another mux γ following it, and let us assume that μ reach a negative mux $\bar{\alpha}$ that should annihilate with α. To continue the reduction, we should be able to split back μ into α and γ. But this would imply to have a way to record into μ the threshold and the names of the ports of γ, that is, to have a structure with the same complexity of the muxes kept disjoint. The key point for the $\vee$ absorption rule is that a $\vee$ node has not a complementary node. The rule corresponding to the propagation of a mux through the auxiliary port of a $\vee$ node is instead definitely unsound. Such a rule, would be sound only in the situation covered by the mux permutation equivalence that we will later introduce studying the reduction properties of the π rules.

Exercise 7.2.3 Define a propagation rule as the bottom one in Figure 7.14, putting a mux in the place of the $\vee$ node. Why the rule cannot be sound? (*Hint:* It would be sound to reverse the direction of the annihilation rule?)

Exercise 7.2.4 A pair of nodes (α_1, α_2) is said a *pathological* redex when α_1 (let m be its threshold) is a mux connected to any port of the logical node α_2 (let i be its index) and $m \geq i$, but $m \neq i$ when α_2 is an @ node and α_1 is connected to its argument port. Prove that no reduct of a λ-tree contains a pathological redex. (*Hint:* Assume that

each edge of a sharing graph has an index and consistently extend the propagation rules to edge indexed redexes. Namely, when a mux crosses an edge, the indexes of the new instances of the edge are lifted by the corresponding offset of the mux. Assign to each edge of a λ-tree the index of the λ-term assigned to it by the decoration rules of Figure 7.8. Prove that the threshold of a mux is always smaller than the index of its principal port edge.)

7.3 Deadlocked redexes

The purpose of the π rules is to get a λ-tree as normal form of the mux propagation. Hence, any situation in which a mux is stuck because it is not part of a redex, and there is no chance that it could be in the future, has to be considered an error.

Definition 7.3.1 (deadlock) A deadlocked π-redex, or a *deadlock* for short, is a pair of nodes (α_1, α_2) as the one in the first item of Definition 7.2.2 that violates the side condition given in the second item of that definition.

A part for the pathological redexes described in Exercise 7.2.4, there are two cases of deadlock:

- A mux pair stuck because the muxes have the same threshold, but their auxiliary ports do not match.
- The mux α_1 is connected to the argument port of the @ node α_2 and the threshold of α_1 is equal to the index of α_2.

Definition 7.3.2 (deadlock freeness) The sharing graph G is said *deadlock-free* when, for any π-reduction $\rho : G \twoheadrightarrow G_1$, the reduct G_1 does not contain any deadlocked π-redex.

The result of Exercise 7.2.4 allows to infer that no reduct of a λ-tree can contain a mux pointing to its root. Therefore, any π normal-form of a deadlock-free sharing graph does not contain muxes at all.

The previous definition does not give any insight on how to characterize deadlock-free sharing graphs: according to it there is no way than reducing them. Moreover, to prove deadlock-freeness we should try all the possible reduction orders, though it is not difficult to find a *standard* reduction strategy reducing λ-trees to λ-trees, simulating step-by-step the usual λ-calculus reduction (see Exercise 7.3.3).

Exercise 7.3.3 (standard strategy) Let us call a sharing graph reduction standard when any of its β-contraction is followed by a maximal sequence of π-rules. In particular, let us write $G \to_s G_1$ when $G \to_\beta G_2 \twoheadrightarrow_\pi G_1$ and G_1 is in π normal-form. Let $T \twoheadrightarrow T_1$ be a λ-term reduction. Prove that $[T] \twoheadrightarrow_s [T_1]$, where $[T]$ and $[T_1]$ are the λ-trees corresponding to T and T_1, respectively.

The algebraic semantics of section 7.7 will give us a characterization of deadlock-freeness for the relevant case in which the π interactions are normalizing. Besides, some useful properties of sharing graphs can be proved taking deadlock-freeness as the only assumption.

7.3.1 Critical pairs

As a consequence of the loss of the uniqueness of the interacting port of a node, the system is no more an interaction net. Then, local confluence is no more for free, but must be proved by case analysis of the critical pairs between π-redexes.

Some critical pairs can be removed assuming that the sharing graph is deadlock-free. For instance, the critical pair depicted in Figure 7.19 (we draw this and the following picture using unary muxes, for the generalization is trivial) may arise in a deadlock-free sharing graph only if $a_1 = a_2$ and $q_1 = q_2$. In which case, after the annihilation of the mux pairs we close the picture getting a commuting diamond.

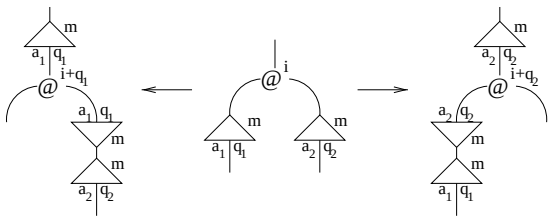


Fig. 7.19. Critical pairs for the π-rules: $m = m_1 = m_2$.

The situation is different when the threshold of the muxes differ, as in Figure 7.20. In this case we have no help from deadlock-freeness. Local confluence is lost even for deadlock-free sharing graph (if we do not assume any other property of the graph).

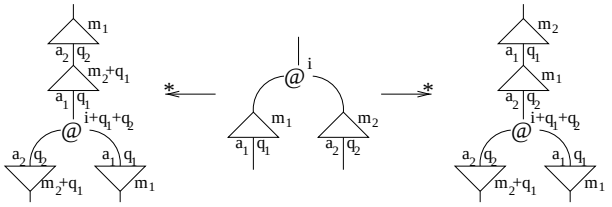


Fig. 7.20. Critical pairs for the π-rules: $m_1 < m_2$.

7.3.2 Mux permutation equivalence

According to the path semantics, the two sharing graphs originated by the critical pair in Figure 7.20 can however be equated modulo some permutation of muxes. Namely, we define $G_1 \sim G_2$, when the sharing graph G_2 is obtainable from G_1 by repeated application of the permutation in Figure 7.21.

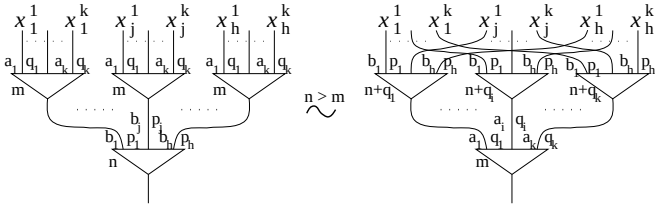
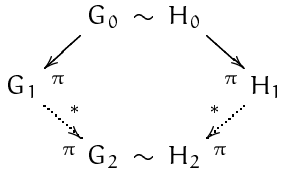


Fig. 7.21. Mux permutation equivalence.

The permutation equivalence ∼ is not part of the π-rules. It has been introduced just to prove the relevant properties of sharing graph reduction. In particular, to show that they are locally confluent modulo ∼ (that means at the same time soundness of permutation equivalence w.r.t. π-reduction).

Lemma 7.3.4 (local confluence) *For any pair of equivalent deadlock-free sharing graphs* $G_0 \sim H_0$*, we have that:*



Proof Let (α_1, α_2) bet the redex of G_0 and (γ_1, γ_2) be the redex of H_0. By definition of $\sim$, there exists a $k \geq 0$ and two sets of trees of muxes $T'_1, \ldots, T'_k \subset G_0$ and $T''_1, \ldots, T''_k \subset H_0$ s.t.: (i) $T'_i \cap T'_j \neq \emptyset$ implies $T'_i = T'_j$ and $T''_i = T''_j$; (ii) $T'_i \sim T''_i$, for $i = 1, \ldots, k$; (iii) $G_0 \setminus (\bigcup_{i=1}^{k} T'_i) = H_0 \setminus (\bigcup_{i=1}^{k} T''_i)$. No more than two of such trees can be involved in the reduction of (α_1, α_2) and (γ_1, γ_2). So, let us assume w.l.o.g. $k = 2$. The prove proceed by case analysis. The relevant cases are: (a) there is an edge connecting the roots of T'_1 and T'_2, and the redexes (α_1, α_2) and (γ_1, γ_2) are the mux pairs composed of the roots of T'_1 and T'_2, and of T''_1 and T''_2, respectively; (b) the muxes α_1 and γ_1 are the roots of T'_1 and T'_2, respectively, and $\alpha_2 = \gamma_2$. In both these cases we see that the trees of muxes can be considered as a sort of big node and that considerations analog to the one pursued for the muxes in Figure 7.20 and Figure 7.21 apply to these big nodes too. □

Exercise 7.3.5 Complete the proof of Lemma 7.3.4.

The previous result shows that we never need to perform any permutation in order to allow π-reduction to proceed. Furthermore, as any π normal-form of a deadlock-free sharing graph is mux free (it is thus the only element of its equivalence class modulo $\sim$), a deadlock-free sharing graph has at most a π normal-form. Using the algebraic semantics, we will also prove that the π-rules are strongly normalizing over sharing graphs, and then the existence and uniqueness of their π normal-form.

Remark 7.3.6 There is no simple way to have something similar to Lemma 7.3.4 in the case of $\pi + \beta$. The problem are the critical pairs between a mux-ν redex and a β-redex. For instance, let us consider the case of a critical pair between an absorption rule and a β-redex. If we would execute the β-rule first we would introduce a mux in front of the one that could be absorbed, and there is no direct way to merge such two muxes. If we would execute the absorption rule first, after the β-rule we would eventually get a sharing graph with a unique mux that is the merging of the two muxes obtained in the other case.

7.4 Sharing morphisms

Given two sharing structures G_1 and G_2, we can assume that they are representations of the same object when G_1 is obtainable by unwinding some sharing contained in G_2, or vice versa. A formal way to express

such a notion is via a partial order relation $G_1 \preccurlyeq G_2$ capturing the intuitive statement: "G_1 is a less-shared-instance of G_2".

According to its intuitive definition, $G_1 \preccurlyeq G_2$ implies the existence of a correspondence by which any node/edge of G_2 can be interpreted as the image of (at least) a node/edge of G_1. Namely, a function mapping the root of G_1 into the root of G_2, a λ node of G_1 into a λ node of G_2, and so on for nodes; plus a function mapping any edge of G_1 connecting the port x of the node α to the port y of the node γ into an edge of G_2 connecting the port x of $M(\alpha)$ to the port y of $M(\gamma)$, with the additional requirement that, when α is a mux and x one of its auxiliary port, the name and offset pairs of the port x of α and of the port x of $M(\alpha)$ coincide.

Definition 7.4.1 (sharing morphism) A mapping M between the edges and the nodes of two graphs G_1 and G_2 is a *sharing morphism*, denoted $M : G_1 \preccurlyeq G_2$, if it is a surjective homomorphism between G_1 and G_2 preserving the type of the nodes, their index/threshold, the name of each port, and the offsets of the auxiliary ports of muxes and $\vee$-nodes.

By the previous definition, only muxes and $\vee$ nodes may change their cardinality under the application of M (remind that two auxiliary ports of a $\vee$ node with the same offset are indistinguishable). The λ and @ nodes have in fact a fixed distinct set of ports. Anyhow, since M is surjective, for any auxiliary port of a mux γ of G_2, there is at least an $\alpha \in M^{-1}(\gamma)$ with a port yielding the same name/offset. Consequently, if G_2 contains a deadlock, also G_1 does. The converse is instead not true. In fact, assume that G_2 contains a k-ary mux annihilating pair. For any function $f = \{1, \ldots, k\} \to \{1, \ldots, k\}$, there is a corresponding way to build a sharing morphism M between the two pairs, but among these, we do not get an immediate deadlock only choosing f equal to the identity.

As a consequence of the previous example, we can infer that sharing morphisms are not yet enough to define the partial order $\preccurlyeq$. Or more precisely that, in order to get a unique completely unshared representation (i.e., containing unary muxes only) of a sharing graph G, we must restrict the class of the sharing graphs, at least to deadlock-free ones.

An example of sharing morphism has been drawn in Figure 7.22. Both the graphs represent $\underline{2} = \lambda x.\, \lambda y.\, (x\ (x\ y))$. If U is the graph on the left and G the one on the right, it is intuitive that U must be a less-shared representation of G, since both the graphs represent the same λ-term

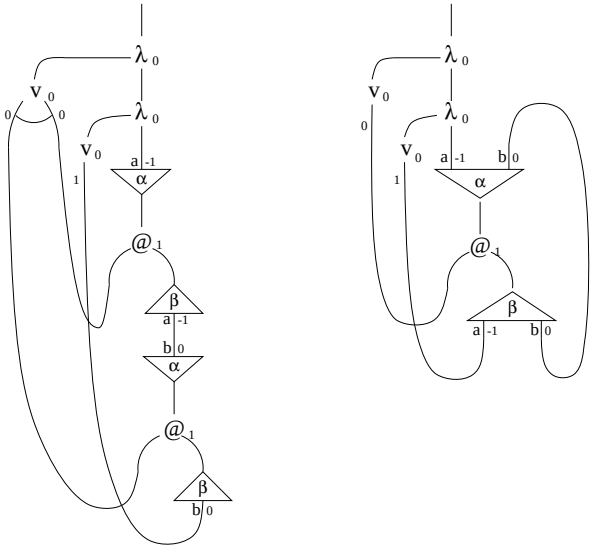


Fig. 7.22. Sharing morphism.

and G is more compact than U; moreover, U is not shared at all, for all the muxes in it are unary. As a matter of fact, $U \preccurlyeq G$ via the sharing morphism mapping the muxes with name α/β in U to the mux with name α/β in G and the two applications in U to the unique application in G (the other correspondences follow by definition of graph morphism).

Even if the muxes in U are unary (also called lifts later) and do not introduce any sharing, there is a tight relation between the unary muxes in U and the binary muxes in G, for the unary muxes mark the boundaries of sharable parts of U. In some sense, the compact representation G corresponds to the superposition of the sharable parts denoted in U by the muxes surrounding the @-nodes. This consideration is crucial for the following analysis. Hidden behind the simulation properties that we will prove in the next sections (Lemma 7.4.3 and Lemma 7.5.4) there is the idea that any reduction of G corresponds to move or create sharing boundaries in some *proper* unshared representation U of G; where proper means that U has some peculiar properties with respect to an arbitrary unfolding of G. In other words, since not all the $U' \preccurlyeq G$ are correct, say proper, the simulation properties are ensured only when we restrict to proper sharing morphisms.

For instance, the graph U′ in Figure 7.23, and many others could be

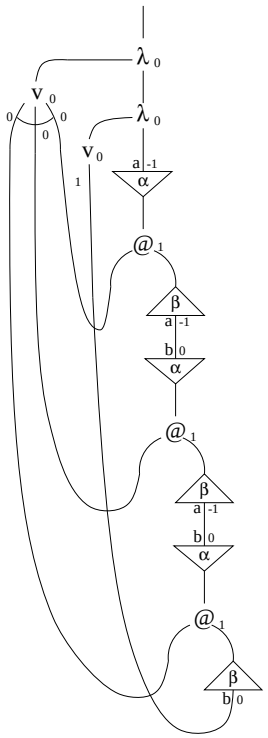


Fig. 7.23. An unsharing of G that is not proper.

given, is obtained by unfolding G; there is in fact a sharing morphism $\mathsf{U}' \preccurlyeq \mathsf{G}$. In spite of this, U' cannot be taken as a proper unsharing of G, for it cannot be a representation of $\underline{2}$ (even if we do not know yet how to associate a λ-term to a sharing graph, at least formally; we can say that the interpretation of G must be $\underline{2}$ for $\mathsf{G} \twoheadrightarrow_\pi \underline{2}$). Moreover, we could easily see that U' is not deadlock-free. In some sense, the boundaries of the sharable parts in U' are inconsistent.

The key point of the algebraic semantics (section 7.7) will be to give the tools for deciding properness, that is, to ensure that in an unshared graph the boundaries of sharable parts are consistent.

Exercise 7.4.2 Verify that both the graphs in Figure 7.22 normalize to the λ-tree of $\underline{2}$ (see Figure 7.24), and that the graph in Figure 7.23 is not deadlock-free.

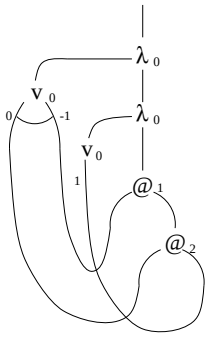


Fig. 7.24. $\underline{2} = \lambda x.\lambda y.(x\,(x\,y))$

7.4.1 Simulation lemma

The study of deadlock-freeness rests on the analysis of the case in which all the muxes are unary. The results obtained for this case will then be generalized exploiting the previously remarked fact that sharing morphisms preserve deadlock-freeness (i.e., G does not contain deadlocked redexes when $M : U \preccurlyeq G$ and U does not contain deadlocked redexes).

First some terminology: a positive (negative) *lift* is a unary (negative) mux; a sharing graph U is said *unshared* if all its muxes are lifts; such a U is said an *unshared instance* of G, if $M : U \preccurlyeq G$.

The relevant point is that any reduction of G corresponds to a reduction of a deadlock-free U, whenever $U \preccurlyeq G$.

Lemma 7.4.3 (simulation: π) *Let U be a deadlock-free unshared graph and let $M : U \preccurlyeq G$. For any $r : G \to_\pi G_1$ there exist a corresponding non-empty reduction $\rho : U \to\!\!\!\!\to_\pi U_1$ and a sharing morphism $M_1 : U_1 \preccurlyeq G_1$. Summarizing in a diagram:*

$$\begin{array}{ccc} & M & \\ U & \preccurlyeq & G \\ {\scriptstyle\rho}\,\vdots\,{\scriptstyle+} & & \downarrow{\scriptstyle r} \\ {\scriptstyle\pi}\downarrow & & {\scriptstyle\pi}\downarrow \\ U_1 & \preccurlyeq & G_1 \\ & M_1 & \end{array}$$

Proof We will show how to build the reduction ρ; then, given ρ, the definition of the sharing morphism M_1 is trivial and it is left as an exercise. Let $r = (\alpha_1, \alpha_2)$, and let (α'_1, α'_2) and (α''_1, α''_2) be two redexes s.t. $M(\alpha'_i) = M(\alpha''_i) = \alpha_i$, for $i = 1, 2$. If $\alpha'_2 = \alpha''_2$ implies $\alpha'_1 = \alpha''_1$,

then $M^{-1}(r)$ does not contain critical pairs and the reduction ρ is any of the possible ordered sequences of the redexes $M^{-1}(r)$. Otherwise, α_2 is a $\vee$ node and α_1 points to one of its auxiliary ports; $\alpha_2' = \alpha_2'' = \gamma$ is a $\vee$ node; and α_1', α_2'' point to two distinct auxiliary ports of γ with the same offset (remind that $\vee$ nodes are the only ones that can have two indistinguishable ports). Furthermore, since U is deadlock-free and α_1' and α_1'' have the same threshold, the auxiliary ports of α_1' and α_1'' have the same name-offset pair (a, q). If r is an absorption, we conclude just absorbing in any order all the muxes in $M^{-1}(\alpha)$. So, let us assume that r is a propagation rule. For any γ s.t. $M(\gamma) = \alpha_2$, let $R_\gamma = \{(\mu, \gamma) \mid M(\mu) = \alpha_1\}$ (by the way, $M^{-1}(r) = \bigcup_\gamma R_\gamma$). Let us use R_γ also to denote the transformation that simultaneously propagates all the lifts μ in the set of redexes R_γ through the $\vee$ node γ. Namely, the reduction R_γ: (*i*) increases by q the index of γ; (*ii*) removes each lift μ directly connecting its auxiliary port to the corresponding auxiliary port of γ; (*iii*) inserts a copy μ' of μ between any auxiliary port of γ that did not interact with a mux μ. For any of such transformation R_γ there is a corresponding π-reduction ρ_γ with the same result. The transformations R_γ do not form critical pairs. Hence, we can take $\rho : U \twoheadrightarrow_\pi U_1$ s.t. U_1 is obtained applying in any order the transformations R_γ, that is, concatenating in any order the reductions ρ_γ. □

In the previous lemma, the reduction of U simulating the reduction of G is not empty. Therefore, whenever the π rules strongly normalize U, they strongly normalize G too.

Proposition 7.4.4 *Let G be a sharing graph with a deadlock-free unshared instance U, i.e., $M : U \preccurlyeq G$ for some sharing morphism M. The sharing graph G is deadlock-free and, when U has no infinite π reduction:*

(i) *the π rules strongly normalize G;*

(ii) *the π normal-form of G is unique and mux free.*

Proof The first item is a direct consequence of Lemma 7.4.3. The second follows from the fact that the unique π normal-form of U does not contain muxes (by the definition of deadlock-freeness and Lemma 7.3.4) and Lemma 7.4.3. □

Exercise 7.4.5 Verify the results proved in this section on the sharing graphs in Figure 7.22. In particular, verify the simulation property of π-rules.

7.5 Unshared beta reduction

Though we have to wait section 7.7 to get a good definition of properness (i.e., independent from the reduction), Proposition 7.4.4 states the minimal requirements that any proper unsharing of G is likely to have. Then, in order to anticipate the relevant results on π-rules, let us define proper sharing graphs via the syntactic properties we would like they hold. Later, in section 7.8.1, we will eventually see that this syntactical definition is equivalent to the semantical one.

Definition 7.5.1 A sharing graph G is *proper* if it is deadlock-free and its π-normal form $\mathcal{R}(\mathsf{U})$ is a λ-tree.

Both the sharing graphs in Figure 7.22 are proper.

Exercise 7.5.2 Let $[\![\,.\,]\!]$ be the partial map that associates a λ-tree T to an unshared graph U by: *(i)* replacing the lifts of U by direct connections between their ports; *(ii)* reindexing T according to the rules in Figure 7.8. (This map is partial since T could not be the abstract syntax tree of a λ-term. In fact, after removing the ν-λ binding connections, we have to check whether T is a tree, and whether the binding connections are correct.) Prove that if U is proper, then $[\![\mathsf{U}]\!] = \mathcal{R}(\mathsf{U})$. (*Hint:* By induction on a normalizing π reduction of U.)

The previous exercise points out that a proper unshared graph U is a λ-tree apart for some lifts and for the indexes of its nodes. The lifts of an unshared instance G have in fact no sharing effect, they rather delimit subgraphs that are shared in G. More precisely, given an unshared graph U representing the λ-tree $\mathsf{T} = [\![\mathsf{U}]\!]$, the lifts of U restrict the possible shared representations of T fixing the points in which sharing may take place (see again the examples in Figure 7.22 and Figure 7.23).

According to the previous syntactical definition, properness is trivially preserved by π-rules; and obviously, this will be the case also for the semantical definition of properness that we will give in section 7.8. Nevertheless, in the semantical approach, this invariance will not be trivial anymore; on the contrary, it will be the key point to be proved.

The next issue is to prove that the β-rule preserves properness. Unfortunately, the proof of this fact is not straightforward and the syntactical definition does not help at all. Indeed, this is the reason because of which we will need a definition of properness that does not mention π-reduction.

In addition, β-rule introduces another obstacle to our plan to study the unshared case first. The point is that unshared graphs are closed under β-reduction only in the case of linear λ-terms. Namely, only if each variable occurs exactly once in the body of its abstraction (remind we are in λI-calculus). Therefore, to get a simulation property similar to the one of Lemma 7.4.3 we have to define an unshared version of β-rule, say β_u-rule, that does not introduce k-ary muxes, see Figure 7.25 (note that, since there is no proviso on the names introduced by a β_u-rule, distinct lifts might have the same name).

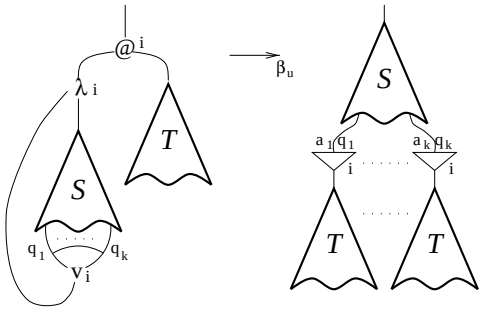


Fig. 7.25. Unshared β-rule.

The β_u-rule splits the reindexing and duplication task of muxes: the duplication in embodied into the rule; the reindexing is instead demanded to the propagation of lifts (see Proposition 7.5.3). This can be seen in more detail looking at the linear case. In that case we would not need any particular implementation of sharing. Nevertheless, the β-rule of Figure 7.10 would introduce a lift to take into account that we have to change the indexes in the tree of the argument of the β-redex. The following propagations of lifts do not cause any duplication, they simply set the right values of that indexes.

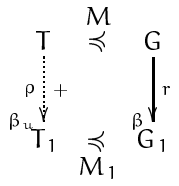
Proposition 7.5.3 *Let* T *be a proper unshared graph. If* $\mathsf{T} \to_{\beta_u} \mathsf{T}'$ *and* T' *is proper, then* $\mathcal{R}(\mathsf{T}) \to_{\beta_\lambda} \mathcal{R}(\mathsf{T}')$.

Proof Using the notation and the result of Exercise 7.5.2, we see that $[\![\mathsf{T}]\!] \to_{\beta_\lambda} [\![\mathsf{T}']\!]$ (by inspection of the β_u-rule) and then that $\mathcal{R}(\mathsf{T}) \to_{\beta_\lambda} \mathcal{R}(\mathsf{T}')$. □

The usefulness of β_u-rule is connected to the proof of soundness of shared β-rule. The key relation between the two rules is established by

the possibility to simulate a β-reduction by a sequence of β_u-reductions in the unshared graph.

Lemma 7.5.4 (simulation: β) *Let T be a proper unshared graph and let $M : T \preccurlyeq G$. For any $r : G \to_\beta G_1$ there exist a corresponding non-empty reduction $\rho : T \to\!\!\!\!\to_{\beta_u} T_1$, and a sharing morphism $M_1 : T_1 \preccurlyeq G_1$. Summarizing in a diagram:*



Proof In the usual way we can define a many-to-one relation s.t. $e \overset{*}{\mapsto} e'$ when e is an edge of T, e' is an edge of T', $T \twoheadrightarrow_{\beta_u} T'$, and e' is a residual of e; and analogously for β_s. Such a relation immediately extend to redexes and, being $R = \{r' \mid M(r') = r\}$ the redexes of T whose image in G is the redex r; the reduction ρ is any complete development of the set of redexes R. (That is, T_1 does not contain any redex r_1 s.t. $r' \overset{*}{\mapsto} r_1$ for some $r' \in R$. It is also easy to see that this definition is sound since β_u has a finite development property similar to the one of β_λ.) For any edge e'_t of T_1 we have a unique edge e_t of T s.t. $e_t \overset{*}{\mapsto} e'_t$. Furthermore, there is a unique edge e'_g of G_1 s.t. $M(e_u) \mapsto e'_g$. Let us take $M_1(e'_u) = e'_g$. It is not difficult to check that M_1 is a sharing morphism. □

We left to prove that β_u preserves properness, but for this we need the algebraic material that we are going to introduce.

7.6 Paths

In Chapter 6 we have seen several notions of paths. The purpose of all of them was to characterize the (virtual) redexes of a sharing graph. Here, as already done for Lamping's proper paths, we restrict to the paths allowing to recover the read-back of a sharing graph G, that is, to the paths connecting the root of G to its v nodes, following the usual syntax tree orientation.

Let $\Phi(U)$ be the set of the oriented paths starting from the root of U and crossing the nodes according to the arrows in Figure 7.26 (the last two nodes are a positive and a negative lift, respectively). Let us assume

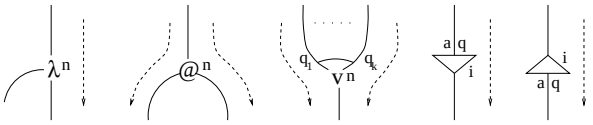


Fig. 7.26. Paths.

that $[\![U]\!]$ is a λ-tree. The unshared graph U is completely characterized by $\Phi(U)$. In fact, any node u of U has an *access path* $\phi e u \in \Phi(U)$, where e is the edge connected to the top port of u (according to the orientation in Figure 7.26). Such an access path is unique for all the nodes but the v ones, which have a path for each occurrence of the corresponding variable—that is, for each input edge. Furthermore, for any occurrence of a variable, there is a cycle $\lambda\psi e\lambda$, where λ is the binder of the variable, e is a binding edge (the edge entering into the binding port of λ), and any path $\chi\lambda\psi' e\lambda$ is a maximal path of $\Phi(U)$ (note that in Figure 7.26 there is no arrow from the binding edge of a λ node to any other edge).

Notation 7.6.1 As already done in the last paragraph, even if a path is a sequence of edges, we will sometimes do explicit some of the nodes crossed by it. In particular, we will use: ϕu to mean that u is the last node of ϕ; $\phi = \psi e$ to mean that e is the last edge of ϕ; $\phi = \psi e u$ to explicit both the last edge and node of ϕ; and so on. The meaning will always be clear since we will use the (indexed) letter e for edges, and the (indexed) letters u, v, @ and λ for the nodes; in particular, λ, @ and v will always denote a node of the corresponding type.

Assuming that in Figure 7.26 the last two nodes are respectively a positive and negative mux, the previous definition of path extends to shared graphs too. Also in this case $\Phi(G)$ is a tree, but, since a sharing graph may contain cycles, $\Phi(G)$ can be infinite (for instance, the sharing graph on the right-hand side of Figure 7.22 has an infinite unfolding). Besides, there is no more a natural isomorphism between the graph and the set of its paths.

Sharing morphisms also relate paths. In fact, given a sharing morphism $M : U \preccurlyeq G$, there is a corresponding path morphism $M : \Phi(U) \preccurlyeq \Phi(G)$. This suggests that in order to find the proper unsharings of G, we need to fix a subset $\Pi(G)$ of the paths $\Phi(G)$—the *proper paths* of G—invariant under sharing morphisms. Namely, $\Pi(G')$ and $\Pi(G)$ are

isomorphic (written $\Pi(G') \simeq \Pi(G)$), whenever $M : G' \preccurlyeq G$. In particular, as it is likely that any path of a proper unshared graph be proper, the natural property of $\Pi(G)$ should be that $\Phi(U) \simeq \Pi(G)$ for any proper unshared instance of G. This corresponds indeed to require the uniqueness of the proper unshared instance of G, because any unshared graph is isomorphic to the set of its paths.

The definition of proper paths (section 7.8) will be given following the techniques used to define Lamping's proper paths in section 3.4. Anyhow, the way in which the dynamic algebra is used here is slightly different. In particular, we have to reformulate some of the algebraic material.

7.7 Algebraic semantics

Let us assume to have an unshared graph U. For any edge of U, we want to find a renaming function by which to compute the actual levels of the nodes of U. The idea is that each node correspond to a relation relate between the renaming functions yielded by the edges incident to it. In particular, the intended interpretation of a lift should then be an operator increasing by its offset the indexes above its threshold.

Definition 7.7.1 (lifting operator) A *lifting operator*, where $m, q, a \in \mathbb{Z}$, and $m \geq 0$, $q \geq -1$, is an indexed transformation $\mathcal{L}[m, q, a]$ of a domain $\mathbb{D}$. The following axiom hold for any m, and any q_i, a_i, d_i, with $i = 1, 2$:

(LO1) $$\mathcal{L}[m, q_1, a_1](d_1) = \mathcal{L}[m, q_2, a_2](d_2)$$
$$\textit{implies} \quad q_1 = q_2 \wedge a_1 = a_2 \wedge d_1 = d_2$$

furthermore, for any $m_1 < m_2$,

(LO2) $$\mathcal{L}[m_2, q_2, a_2]\,\mathcal{L}[m_1, q_1, a_1] = \mathcal{L}[m_1, q_1, a_1]\,\mathcal{L}[m_2 + q_1, q_2, a_2]$$

(LO3) $$\mathcal{L}[m_1, q_1, a_1](d_1) = \mathcal{L}[m_2, q_2, a_2](d_2)$$
$$\textit{iff}$$
$$\exists\, d :\ \mathcal{L}[m_2 + q_1, q_2, a_2](d) = d_1 \wedge \mathcal{L}[m_1, q_1, a_1](d) = d_2$$

The index m is the (lifting) *threshold* of $\mathcal{L}[m, q, a]$, the index q is its (lifting) *offset*, and the index a is its *port name*.

Lifting operators are the intended interpretation of lifts. The idea is that any edge of an unshared graph U is in the scope of a set of lifts: the ones that should cross the edge normalizing U. Assuming that lifting

operators give a compositional semantics for lifts, we expect that the reindexing operator associated to an edge be a product of them, say a *lifting sequence*, in which there is a factor for any lift containing the edge in its scope.

Before to go on, let us note that lifts, and then lifting operators, are indeed something more than renaming functions. The first reason is that we have to keep track of their matching. The second problem is that, assuming to move along a path, at a negative lift we have to restore the names that where in act at the matching positive lift; more formally, a lifting operator must be at least left invertible.

To clarify the latter point let us try to interpret a lifting operators as integer functionals. Namely, let us assume $\mathbb{D} = \mathbb{Z}^{\mathbb{Z}}$ and

$$[\![\mathcal{L}[m, q, a]]\!] = \delta_{m,q}$$

where

$$\delta_{m,q}(f)(i) = \begin{cases} f(i) & \text{when } i \leq m \\ f(i+q) & \text{otherwise} \end{cases}$$

The integer lifting functionals have a commutative property analog to the one of lifting operators in axioms (LO2) and (LO3).

Exercise 7.7.2 Prove that, for any $q_1, q_2 \geq -1$,

$$\delta_{m_2,q_2}\delta_{m_1,q_1} = \delta_{m_1,q_1}\delta_{m_2+q_1,q_2}$$

and that

$$\delta_{m_1,q_1}(f) = \delta_{m_2,q_2}(g) \quad \text{iff} \quad \exists(h)(\delta_{m_1,q_1}(h) = g \wedge \delta_{m_2+q_1,q_2}(h) = f)$$

when $m_1 < m_2$.

One of the reasons because of which such an interpretation is inadequate is that axiom (LO1) fails, that is, we do not have a correct algebraic encoding of the annihilation rule, for the functional $\delta_{m,q}$ is not injective. Such a naive interpretation, even if not completely satisfactory, is however helpful for the intuitive understanding of the behavior of lifts. For instance, a lifting integer functional compress the names between m and $m+q$ (let us assume $q \geq 0$) into the range m and $m+1$ and, since we are in $\mathbb{Z}^{\mathbb{Z}}$, the values internal to the range are lost.

The latter reasoning on the range in which a lifting operator acts extends to lifting sequences too. Unfortunately, the corresponding formalization requires a rather technical condition, we hope that some explanation on its meaning will be given by Exercise 7.7.4.

Definition 7.7.3 A *lifting sequence* from n_0 to n_1, with $n_0 \leq n_1$, is a product of lifting operators

$$\mathcal{H} = \prod_{0<i\leq k} \mathcal{L}[m_i, q_i, a_i]$$

in which, for $i = 1, 2, \ldots, k$,

$$n_0 \leq m_i < n_1 + \sum_{0<j<i} q_j.$$

The set $\mathsf{LSeq}[n_0, n_1]$ is the family of the lifting sequences from n_0 to n_1.

In particular, assuming $n_1 = \omega$ we get the family $\mathsf{LSeq}[n_0, \omega]$ of the lifting sequences $\mathcal{H}$ for which $m_i \geq n_0$. Then, $\mathsf{LSeq} = \mathsf{LSeq}[0, \omega]$ is the family of all the lifting sequence. By inspection of the definition, we also see that $\mathsf{LSeq}[n_0, n_1] \subseteq \mathsf{LSeq}[m_0, m_1]$, when $m_0 \leq n_0 \leq n_1 \leq m_1$. Therefore, if we want to characterize the range of indexes on which a lifting operator $\mathcal{F}$ works we have to choose n_0 and n_1 s.t. $\mathcal{F} \in \mathsf{LSeq}[m_0, m_1]$ implies $m_0 \leq n_0$ and $m_1 \geq n_1$.

Exercise 7.7.4 Let $\mathcal{F} \in \mathsf{LSeq}[n_0, n_1]$. If Q is the sum of the offsets of the lifting operators in $\mathcal{F}$, prove that

$$[\![\mathcal{F}]\!](f)(i) = \begin{cases} f(i) & \text{when } i \leq n_0 \\ f'(i) & \text{when } n_0 < i < n_1 \\ f(i+Q) & \text{when } i \geq n_1 \end{cases}$$

for some $f' \in \mathbb{Z}^{\mathbb{Z}}$.

7.7.1 Left inverses of lifting operators

Lifting sequences are injective transformations of $\mathbb{D}$: they are the composition of injective operators (remind axiom (LO1)). Hence, each lifting sequence $\mathcal{F}$ has a non unique left inverse $\mathcal{F}^*$ that may assume any value outside of the codomain of $\mathcal{F}$. Anyhow, if we consider partial functions too, the natural choice for $\mathcal{F}^*$ is the less defined partial transformation s.t. $\mathcal{F}\mathcal{F}^*\mathcal{F} = \mathcal{F}$, i.e., s.t. $\mathsf{dom}(\mathcal{F}^*) = \mathsf{codom}(\mathcal{F})$ and $\mathcal{F}^*(\mathcal{F}(d)) = d$, for any $d \in \mathsf{dom}(\mathcal{F})$.

In particular, defined

$$\overline{\mathcal{L}}[m, q, a](d) = \begin{cases} \bar{d} & \text{when } d = \mathcal{L}[m, q, a](\bar{d}) \\ \bot & \text{when } d \notin \mathsf{codom}(\mathcal{L}[m, q, a]) \end{cases}$$

where $\overline{\mathcal{L}}[\mathfrak{m},\mathfrak{q},\mathfrak{a}](\mathfrak{d}) = \perp$ is just a denotation for $\mathfrak{d} \notin \mathsf{dom}(\mathcal{L}[\mathfrak{m},\mathfrak{q},\mathfrak{a}])$, we have

$$\begin{aligned}
\mathcal{L}[\mathfrak{m},\mathfrak{q},\mathfrak{a}]^* &= \overline{\mathcal{L}}[\mathfrak{m},\mathfrak{q},\mathfrak{a}] \\
\widetilde{\mathcal{L}}[\mathfrak{m},\mathfrak{q},\mathfrak{a}] &= (\widetilde{\mathcal{L}}[\mathfrak{m},\mathfrak{q},\mathfrak{a}]^*)^* \\
(\widetilde{\mathcal{L}}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1]\cdots\widetilde{\mathcal{L}}[\mathfrak{m}_k,\mathfrak{q}_k,\mathfrak{a}_k])^* &= \widetilde{\mathcal{L}}[\mathfrak{m}_k,\mathfrak{q}_k,\mathfrak{a}_k]^*\cdots\widetilde{\mathcal{L}}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1]^*
\end{aligned}$$

where $\widetilde{\mathcal{L}}[\mathfrak{m}_i,\mathfrak{q}_i,\mathfrak{a}_i]$ stands for either $\mathcal{L}[\mathfrak{m}_i,\mathfrak{q}_i,\mathfrak{a}_i]$ or $\overline{\mathcal{L}}[\mathfrak{m}_i,\mathfrak{q}_i,\mathfrak{a}_i]$

Lifting operators and their left inverses form a monoid of injective partial transformations of $\mathbb{D}$ closed under left inversion. Furthermore, the axioms of the lifting operators can be reformulated using the left inverse operation.

Exercise 7.7.5 Prove that the axioms of Definition 7.7.1 are equivalent to the axioms:

$$\begin{aligned}
\overline{\mathcal{L}}[\mathfrak{m},\mathfrak{q},\mathfrak{a}]\,\mathcal{L}[\mathfrak{m},\mathfrak{q},\mathfrak{a}] &= 1 \\
\overline{\mathcal{L}}[\mathfrak{m},\mathfrak{q}_2,\mathfrak{a}_2]\,\mathcal{L}[\mathfrak{m},\mathfrak{q}_1,\mathfrak{a}_1] &= 0
\end{aligned}$$

for any $\mathfrak{m},\mathfrak{q}$, and $\mathfrak{q}_1 \neq \mathfrak{q}_2$ or $\mathfrak{a}_1 \neq \mathfrak{a}_2$;

$$\begin{aligned}
\mathcal{L}[\mathfrak{m}_2,\mathfrak{q}_2,\mathfrak{a}_2]\,\mathcal{L}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1] &= \mathcal{L}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1]\,\mathcal{L}[\mathfrak{m}_2+\mathfrak{q}_1,\mathfrak{q}_2,\mathfrak{a}_2] \\
\overline{\mathcal{L}}[\mathfrak{m}_2,\mathfrak{q}_2,\mathfrak{a}_2]\,\mathcal{L}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1] &= \mathcal{L}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1]\,\overline{\mathcal{L}}[\mathfrak{m}_2+\mathfrak{q}_1,\mathfrak{q}_2,\mathfrak{a}_2]
\end{aligned}$$

for any $\mathfrak{q}_1,\mathfrak{q}_2$, $\mathfrak{a}_1,\mathfrak{a}_2$, when $\mathfrak{m}_1 < \mathfrak{m}_2$.

By the way, also the dual properties of the last permutation rules hold:

$$\begin{aligned}
\overline{\mathcal{L}}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1]\,\mathcal{L}[\mathfrak{m}_2,\mathfrak{q}_2,\mathfrak{a}_2] &= \mathcal{L}[\mathfrak{m}_2+\mathfrak{q}_1,\mathfrak{q}_2,\mathfrak{a}_2]\,\overline{\mathcal{L}}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1] \\
\overline{\mathcal{L}}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1]\,\overline{\mathcal{L}}[\mathfrak{m}_2,\mathfrak{q}_2,\mathfrak{a}_2] &= \overline{\mathcal{L}}[\mathfrak{m}_2+\mathfrak{q}_1,\mathfrak{q}_2,\mathfrak{a}_2]\,\overline{\mathcal{L}}[\mathfrak{m}_1,\mathfrak{q}_1,\mathfrak{a}_1]
\end{aligned}$$

where $\mathfrak{m}_1 < \mathfrak{m}_2$.

7.7.2 The inverse semigroup LSeq^*

Let us remind the definition and the main property of inverse semigroups (see section 6.4.2). An *inverse semigroup* M with a null element 0 s.t. $0\mathcal{F} = 0 = \mathcal{F}0$, for any $\mathcal{F}$, is a monoid closed under an involution operation $(\cdot)^*$ s.t., for any $\mathcal{F},\mathcal{F}_1,\mathcal{F}_2 \in \mathsf{M}$,

$$\begin{aligned}
\mathcal{F}^{**} &= \mathcal{F} \\
(\mathcal{F}_1\mathcal{F}_2)^* &= \mathcal{F}_2^*\mathcal{F}_1^* \\
\langle\mathcal{F}\rangle\mathcal{F} &= \mathcal{F} \\
\langle\mathcal{F}_1\rangle\langle\mathcal{F}_2\rangle &= \langle\mathcal{F}_2\rangle\langle\mathcal{F}_1\rangle
\end{aligned}$$

where $\langle \mathcal{F} \rangle \equiv \mathcal{F}\mathcal{F}^*$.

From these axioms we can prove that $\mathcal{F}$ is an idempotent iff $\mathcal{F} = \langle \mathcal{F} \rangle$ and that for any idempotent $\mathcal{F}$ we have $\mathcal{F} = \mathcal{F}^*$. That in particular implies $1^* = 1$ and $0^* = 0$.

Definition 7.7.6 (LSeq^*) The inverse semigroup LSeq^* is the smallest inverse semigroup generated by the family of the indexed symbols $\mathcal{L}[m, q, a]$ and of their left inverse symbols $\overline{\mathcal{L}}[m, q, a]$ (being $m \geq 0$ and $q \geq -1$) according to the axioms

(LS0) $$\mathcal{L}[m, q, a]^* = \overline{\mathcal{L}}[m, q, a]$$

(LS1) $$\overline{\mathcal{L}}[m, q, a]\, \mathcal{L}[m, q, a] = 1$$

(LS1) $$\overline{\mathcal{L}}[m, q_2, a_2]\, \mathcal{L}[m, q_1, a_1] = 0 \qquad \text{if } q_1 \neq q_2 \text{ or } a_1 \neq a_2$$

(LS3) $$\mathcal{L}[m_2, q_2, a_2]\, \mathcal{L}[m_1, q_1, a_1] = \mathcal{L}[m_1, q_1, a_1]\, \mathcal{L}[m_2 + q_1, q_2, a_2]$$

(LS4) $$\overline{\mathcal{L}}[m_2, q_2, a_2]\, \mathcal{L}[m_1, q_1, a_1] = \mathcal{L}[m_1, q_1, a_1]\, \overline{\mathcal{L}}[m_2 + q_1, q_2, a_2]$$

when $m_1 < m_2$.

The monoid LSeq is instead the smallest monoid $\mathsf{LSeq} \subset \mathsf{LSeq}^*$ containing the indexed symbols $\mathcal{L}[m, q, a]$ (for any $m \geq 0$ and $q \geq -1$) and closed by composition.

By the way, the intended interpretation of LSeq^* is the monoid generated by the lifting operators and by their left inverse partial functions. In fact, it is readily seen that:

(i) The constant 0 is the nowhere defined partial transformation of $\mathbb{D}$.
(ii) Any idempotent $\langle \mathcal{F} \rangle$ of LSeq^* is the identity map restricted to the codomain of $\mathcal{F}$, i.e., $\mathsf{dom}(\langle \mathcal{F} \rangle) = \mathsf{codom}(\mathcal{F})$.
(iii) For any $\mathcal{H} \in \mathsf{LSeq}$ we have that $\mathcal{H}^*\, \mathcal{H} = \langle \mathcal{H}^* \rangle = 1$, since $\mathsf{codom}(\mathcal{H}^*) = \mathsf{dom}(\mathcal{H}) = \mathbb{D}$.

Also in this case we have an AB^* theorem.

Lemma 7.7.7 *For any $\mathcal{F} \in \mathsf{LSeq}^*$, $\mathcal{F} \neq 0$, there exist $\mathcal{H}_+, \mathcal{H}_- \in \mathsf{LSeq}$ s.t. $\mathcal{F} = \mathcal{H}_+ \mathcal{H}_-^*$.*

Proof By induction on $|\mathcal{F}|$. The case $|\mathcal{F}| = 0$ and the case $\mathcal{F} = \mathcal{L}[m, q, a]\, \mathcal{F}_1$ (just apply the induction hypothesis) are direct. So, let us take $\mathcal{F} = \overline{\mathcal{L}}[m, q, a]\, \mathcal{F}_1$. By induction hypothesis, we get $\mathcal{F} = \overline{\mathcal{L}}[m, q, a]\, \mathcal{H}_+ \mathcal{H}_-^*$, with $\mathcal{H}_+, \mathcal{H}_- \in \mathsf{LSeq}$. Let $\mathcal{H}_+ = \mathcal{L}[m_1, q_1, a_1] \cdots \mathcal{L}[m_k, q_k, a_k]$

be in canonical form, and let h be the first index for which $m+Q_i \geq m_h$ (where $Q_i = \sum_{i<h} q_i$), or let $h = k+1$ when such an index does not exist. Let $\mathcal{H} = \mathcal{H}_0\mathcal{H}_1$ with $\mathcal{H}_0 = \mathcal{L}[m_1, q_1, a_1] \cdots \mathcal{L}[m_{h-1}, q_{k-1}, a_{k-1}]$. If $m_h = m+Q_i$, we have either $q_h = q$ and $a_h = a$, or not. In the first case $\overline{\mathcal{L}}[m, q, a]\mathcal{H}_+ \in \mathsf{LSeq}$, in the second $\overline{\mathcal{L}}[m, q, a]\mathcal{H}_+ = 0$ and then $\mathcal{H} = 0$. When $m_h > m+Q_i$ instead, we see that $\mathcal{H} = \mathcal{H}_0\mathcal{H}_1^{+q}\overline{\mathcal{L}}[m+Q_i, q, a]\,\mathcal{H}_-^*$, where $\mathcal{H}_1^{+q}$ is the lifting sequence obtained increasing by q all the thresholds in $\mathcal{H}_1$. □

The elements of LSeq^* can then be written in *canonical form*. In more detail, Axiom (LS3) allows to state that any lifting sequence can be written in a canonical way by arranging it in a product of lifting operators whose thresholds are non-decreasingly ordered. Therefore, $\mathcal{F} \equiv \mathcal{H}_+\mathcal{H}_-^*$ is in canonical form only whenever both $\mathcal{H}_+$ and $\mathcal{H}_-$ are canonical.

Lemma 7.7.8 *Let $\mathcal{H}_+, \mathcal{H}_- \in \mathsf{LSeq}$. We have that $\mathcal{H}_+\,\mathcal{H}_-^* = 1$ iff $\mathcal{H}_+ \equiv \mathcal{H}_- \equiv 1$.*

Proof Let us assume $\mathcal{H}_+ = \mathcal{L}[m, q, a]\,\mathcal{H}'$. For any pair q', a' s.t. $q \neq q'$ or $a \neq a'$. We would get $1 = \overline{\mathcal{L}}[m, q', a']\,\mathcal{H}_+\,\mathcal{H}_-^*\,\mathcal{L}[m, q', a'] = 0$. Hence, $\mathcal{H}_+ \equiv 1$, etc. □

Remark 7.7.9 In the latter lemma we implicitly assumed $1 \neq 0$. More precisely we should have said that the only model in which Lemma 7.7.8 does not hold is the trivial one. The same reasoning apply to the uniqueness of the canonical form proved by the next proposition.

Lemma 7.7.10 *For any pair $\mathcal{H}_1, \mathcal{H}_2 \in \mathsf{LSeq}$, $\langle\mathcal{H}_1\rangle = \langle\mathcal{H}_2\rangle$ iff $\mathcal{H}_1 = \mathcal{H}_2$.*

Proof Let $\mathcal{H}_+\,\mathcal{H}_-^*$ be a canonical form of $\mathcal{H}_1^*\,\mathcal{H}_2$. We have $1 = \mathcal{H}_1^*\langle\mathcal{H}_1\rangle\mathcal{H}_1 = \mathcal{H}_1^*\langle\mathcal{H}_2\rangle\mathcal{H}_1 = \mathcal{H}_1^*\,\mathcal{H}_2\,(\mathcal{H}_1^*\,\mathcal{H}_2)^* = \mathcal{H}_+\,\mathcal{H}_-^*\,\mathcal{H}_-\,\mathcal{H}_+^* = \mathcal{H}_+\mathcal{H}_+^*$. From which, we see (by Lemma 7.7.8) that $\mathcal{H}_+ = 1$. In a similar way we see that $\mathcal{H}_- = 1$ and then that $\mathcal{H}_1^*\,\mathcal{H}_2 = 1$. Hence, $\mathcal{H}_2 = \langle\mathcal{H}_2\rangle\,\mathcal{H}_2 = \langle\mathcal{H}_1\rangle\,\mathcal{H}_2 = \mathcal{H}_1$. □

Proposition 7.7.11 *Two elements of LSeq^* are equal iff their canonical forms coincide.*

Proof Let $\mathcal{H}_+\mathcal{H}_-^*$ and $\widehat{\mathcal{H}}_+\widehat{\mathcal{H}}_-^*$ be two canonical forms of $\mathcal{F} \in \mathsf{LSeq}^*$. We have $\langle \mathcal{H}_+ \rangle = \langle \mathcal{F} \rangle = \langle \widehat{\mathcal{H}}_+ \rangle$. Then, by Lemma 7.7.10 $\mathcal{H}_+ = \widehat{\mathcal{H}}_+$. From which, $\mathcal{H}_- = \widehat{\mathcal{H}}_-$ too. Hence, we may restrict to show the uniqueness of the canonical form of a lifting sequence. The case of 1 is proved by Fact 7.7.8. So, let us assume that $\mathcal{L}[m_1, q_1, a_1]\,\mathcal{H}_1 = \mathcal{L}[m_2, q_2, a_2]\,\mathcal{H}_2$ are both in canonical form and, w.l.o.g., that $m_1 \leq m_2$. If $m_1 < m_2$, then $\mathcal{H}_1 = \mathcal{L}[m_2 + q_1, q_2, a_2]\,\mathcal{H}_2^{+q_1}\overline{\mathcal{L}}[m_1, q_1, a_1]$, that leads to the contradiction $\mathcal{H}_1\mathcal{L}[m_1, q_1', a_1'] = 0$ when $q_1' \neq q_1$ or $a_1' \neq a_1$. Hence, the only possibility is $m_1 = m_2$, and then $q_1 = q_2$ and $a_1 = a_2$ too. Summarizing, from the initial canonical forms we have got two shorter equivalent ones $\mathcal{H}_1 = \mathcal{H}_2$. So, by induction hypothesis, we see that $\mathcal{H}_1$ and $\mathcal{H}_2$ coincide, etc. □

7.8 Proper paths

Let us now proceed with the original goal to associate to each edge the reindexing operator corresponding to the muxes that have to cross it. In this way, we will give two new equivalent definitions of proper sharing graphs based on the algebraic semantics. Eventually, by Proposition 7.8.15, we will also show that semantical properness coincides with the syntactical definition of proper sharing graphs that we gave in Definition 7.5.1.

A *lifting sequence assignment* is a map associating to each edge of an unshared graph a weight $\mathcal{F} \in \mathsf{LSeq}$ according to the constraints in Figure 7.27. In particular, since the first node of $\cup$ must always have an index 0, the constraint on $\mathcal{F}$ implies that the lifting sequence assigned to the root is always 1.

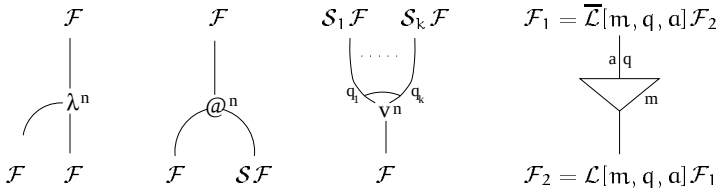


Fig. 7.27. In all the first three cases, $\mathcal{F} \in \mathsf{LSeq}[0, n]$. In the @-node case, $\mathcal{S} \in \mathsf{LSeq}[n, n+1]$. In the ∨-node case $\mathcal{S}_i \in \mathsf{LSeq}[n, n+q_i+1]$, for $i = 1, \ldots, k$.

There are unshared graphs for which there is no lifting sequence assignment (e.g., the unshared graph in Figure 7.23).

Exercise 7.8.1 Prove that an unshared graph containing a deadlock does not admit any lifting sequence assignment.

Exercise 7.8.2 Verify that the sharing graphs in Figure 7.22 are proper (according to the previous semantical definition).

According to the result of the exercise, lifting sequence assignments are likely to be the algebraic characterization of deadlock-freeness. Anyhow, before to prove that—by showing that the existence of an assignment is invariant under unshared reduction—let us analyze in more details how to constructively check whether an unshared graph $\mathcal{U}$ admits a lifting assignment or not.

The constraints in Figure 7.27 can be oriented according to the orientation of the paths in the tree $\Phi(\mathcal{U})$ (this means that the last constraint in Figure 7.27 has to be read top-down in the case of a positive lift and bottom-up in the case of a negative one). We get in this way a functional dependency between the weights of the edges corresponding to the descendant relation of $\Phi(\mathcal{U})$. The only cause of indeterminacy remaining the parameters $\mathcal{S}$ in the equations associated to the @-nodes.

A *state* σ of $\mathcal{U}$ is a map associating a lifting sequence $\mathcal{S}_u \in \mathsf{LSeq}[n, n+1]$ to each @-node u of $\mathcal{U}$, where n is the index of u. A state σ is *compatible* if there is a lifting sequence assignment s.t. the constraints of each @-node u are satisfied replacing $\mathcal{S}_u$ for $\mathcal{S}$.

The weight of any edge of $\mathcal{U}$ is completely determined once fixed a compatible state σ. Moreover, a simple visit (in any order) of the tree $\Phi(\mathcal{U})$ from its root to its v-nodes gives a constructive procedure to decide if a state is compatible.

Let us be more precise. We start assigning at the root the lifting sequence $\mathsf{1}$, and we proceed according to the equations in Figure 7.27. This first part of the procedure may fail only reaching the principal port of a negative lift with a weight $\mathcal{H}$ s.t. $\overline{\mathcal{L}}[m, q, a]\mathcal{H} \notin \mathsf{LSeq}$, where m is the threshold of the lift, q its offset, and a its port name. In other words, because of the properties of lifting sequences, the weight associated to the principal port edge of a negative lift must be $\mathcal{L}[m, q, a]\mathcal{H}$. Let us remark that the latter requirement is the algebraic encoding of the property that for each negative control node there must be a matching positive node.

Assuming that we do not fail because of a negative lift, we left to check the constraints on the v-nodes. At first glance, it might seem that this constraints are a source of indeterminacy: for any edge entering into a v-node, we have a lifting sequence $\mathcal{H}$ that we want to split in the

product of two sequences $\mathcal{S} \in \mathsf{LSeq}[n, n+q+1]$ and $\mathcal{F} \in \mathsf{LSeq}[0, n]$, where q is the index of the edge and n the index of the node. The next lemma ensures instead that the previous decomposition, if any, is indeed unique.

Lemma 7.8.3 *Let n be an index s.t. $n_0 \leq n \leq n_1$. Given $\mathcal{H} \in \mathsf{LSeq}[n_0, n_1]$, there is a unique sectioning of $\mathcal{H}$ (w.r.t. to the index n) in two lifting sequences $\mathcal{H}^{<n} \in \mathsf{LSeq}[n_0, n]$ and $\mathcal{H}^{\geq n} \in \mathsf{LSeq}[n, n_1]$ s.t. $\mathcal{H} = \mathcal{H}^{\geq n}\mathcal{H}^{<n}$.*

Proof Let $\mathcal{H} = \mathcal{L}[m_1, q_1, a_1] \cdots \mathcal{L}[m_k, q_k, a_k] \in \mathsf{LSeq}[n_1, n_2]$ be in canonical form. Let h be the first index for which $n + Q_i \geq m_i$ (where $Q_i = \sum_{i<h} q_i$), if any, otherwise let $h = k+1$. Let us take

$$\mathcal{H}^{<n} = \mathcal{L}[m_1, q_1, a_1] \cdots \mathcal{L}[m_{h-1}, q_{k-1}, a_{k-1}]$$

and

$$\mathcal{H}^{\geq n} = \mathcal{L}[m_h - Q_h, q_h, a_h] \cdots \mathcal{L}[m_k - Q_h, q_k, a_k].$$

By definition, $\mathcal{H}^{<n} \in \mathsf{LSeq}[0, n]$ and $\mathcal{H}^{\leq n} \in \mathsf{LSeq}[n, \omega]$. The definition of $\mathcal{H}^{<n}$ and $\mathcal{H}^{\geq n}$ also proves their uniqueness. □

Incidentally, let us note that the previous lemma also implies that a decomposition $\mathcal{H} = \mathcal{S}\mathcal{F}$, with $\mathcal{S} \in \mathsf{LSeq}[n, n+q+1]$ and $\mathcal{F} \in \mathsf{LSeq}[0, n]$, is possible if and only if $\mathcal{H} \in \mathsf{LSeq}[0, n+q+1]$.

After the assignment phase, we can thus determine the value of $\mathcal{F}$ for any edge entering into a ν-node. Only when each of the previous values is equal to the lifting sequence associated to the λ-node binding the corresponding variable, we can finally say that the lifting sequence assignment is correct. (We invite the reader to compare this requirement, with the transparency property used proving correctness of Lamping's algorithm in section 3.5.1.)

Definition 7.8.4 A sharing graph G is *proper* if there exists an unshared graph U and a sharing morphism M s.t. $M : U \preccurlyeq G$ and U admits a lifting sequence assignment for each internal state σ.

Any proper unshared graph U s.t. there exists $M : U \preccurlyeq G$ is said a *complete unsharing* of G. In particular, an unshared graph that admits an assignment for any state is proper and is the complete unsharing of itself.

Exercise 7.8.5 Prove that any λ-tree is a proper sharing graph.

Assignments and proper sharing graphs may be studied also in terms of paths. From that point of view, the constraints in Figure 7.27 become the rules to associate a transfer function $\mathcal{F}_\phi[\sigma] \in \mathsf{LSeq}^*$, dependent on the state σ, to each path ϕ.

Denoted by $\Pi(\mathsf{U})$ the biggest subset of $\Phi(\mathsf{U})$ s.t. $\mathcal{F}_\phi[\sigma] \in \mathsf{LSeq}$ for any state σ, an equivalent definition of proper unshared graph is:

(i) $\Pi(\mathsf{U}) = \Phi(\mathsf{U})$.
(ii) for any $\phi \nu e \lambda \in \Pi(\mathsf{U})$, if $\phi = \chi\lambda\psi\nu$, there exists $\mathcal{S}[\sigma] \in \mathsf{LSeq}[\mathfrak{n}, \mathfrak{n}+\mathfrak{q}+1]$, s.t. $\mathcal{F}_\phi[\sigma] = \mathcal{S}[\sigma]\mathcal{F}_\chi[\sigma]$, where $\mathfrak{n}$ is the level of λ and $\mathfrak{q}$ the offset of the port of the ν node reached by ϕ.

The advantage of the latter approach is that the definition of properness can be reformulated for all the sharing graphs (not only the unshared ones), extending the notation in the natural way.

Definition 7.8.6 The sharing graph G is *proper* if:

(i) $\Pi(\mathsf{G})$ spans G.
(ii) Any maximal path $\phi' \in \Pi(\mathsf{G})$ is finite and ends at the binding port of a λ-node, that is, $\phi' = \phi e \lambda$.
(iii) For any maximal path $\phi e \lambda \in \Pi(\mathsf{G})$, there in an occurrence of λ in ϕ s.t $\phi = \chi\lambda\psi$ and, for any state σ, there exists $\mathcal{S}[\sigma] \in \mathsf{LSeq}[\mathfrak{n}, \mathfrak{n}+\mathfrak{q}+1]$, s.t. $\mathcal{F}_\phi[\sigma] = \mathcal{S}[\sigma]\mathcal{F}_\chi[\sigma]$, where $\mathfrak{n}$ is the level of λ and $\mathfrak{q}$ the offset of the port reached by ϕ.

Let us note that the second proviso of this definition does not implies $\mathcal{F}_\psi[\sigma] \in \mathsf{LSeq}[\mathfrak{n}, \mathfrak{n}+\mathfrak{q}+1]$. In fact, we might have $\mathcal{F}_\psi[\sigma] = \mathcal{S}\mathcal{H}\mathcal{H}^*$, for some $\mathcal{H} \in \mathsf{LSeq}$ s.t. $\mathcal{F}_\chi[\sigma] = \mathcal{H}\mathcal{H}_0$.

To see that Definition 7.8.4 and Definition 7.8.6 are equivalent let us start noticing that the *proper paths* $\Pi(\mathsf{G})$ of a proper sharing graph G are the proper paths of its complete unsharing.

Lemma 7.8.7 *If* U *is a complete unsharing of* G*, then* $\Pi(\mathsf{U}) \simeq \Pi(\mathsf{G})$.

Proof Let $\mathsf{M} : \mathsf{U} \preccurlyeq \mathsf{G}$. Since $\mathcal{F}_\phi = \mathcal{F}_{\mathsf{M}(\phi)}$, we have $\mathsf{M}(\phi) \in \Pi(\mathsf{G})$, for any $\phi \in \Pi(\mathsf{U})$. Vice versa, to any $\psi \in \Pi(\mathsf{G})$ we prove that there is a path $\phi \in \Pi(\mathsf{U})$, with $\mathsf{M}(\phi) = \psi$, by induction on the length of ψ. The only relevant case is when $\psi = \psi' e \mathfrak{u}$ and the last node of ψ' is a negative mux. By the induction hypothesis, there is $\phi' \in \Pi(\mathsf{U})$ s.t. $\mathsf{M}(\phi') = \psi'$. Since U is a proper unshared graph (according to Definition 7.8.4) and $\mathfrak{u}$

is a negative mux (hence, ϕ' ends at the principal port of a negative lift) there is a unique path $\phi = \phi' e' u' \in \Pi(G)$, $\mathcal{F}_\phi = \overline{\mathcal{L}}[m, q, a]\mathcal{F}_{\phi'} \in \mathsf{LSeq}$ (where m, q, a are the parameters of the lift), and $M(\phi) \in \Pi(G)$. We show that $M(\phi) = \psi$. In fact, since $\mathcal{L}[m, q, a]\mathcal{F}_\phi = \mathcal{F}_{\phi'} = \mathcal{F}_{\psi'}$, the edge e must be connected to the port with name a of the negative mux at the end of ψ'. Thus, $M(e') = e$ and $M(\phi) = \psi$. □

The main consequence of the previous lemma is that any proper sharing graph according to Definition 7.8.4 is proper also according to Definition 7.8.6. For the converse, let us note that the requirements in Definition 7.8.6 allow to associate a proper unshared graph to $\Pi(G)$, and then to G. In fact, interpreting a path $\phi \in \Pi(G)$ ending at a node u as a node of the same type of u , it is natural to see $\Pi(G)$ as an unfolding tree of G. The last proviso of Definition 7.8.6 suggests then how to close the tree in order to get a proper unshared graph U.

Let us be more precise. Any path $\phi e u \in \Pi(G)$ s.t. e is not a binding connection and u is not a v-node is an internal node of U with the same type and index of u (a lift with the same threshold of u and the appropriate threshold, port name, and offset in the case in which u is a mux). It is easy to check that any lift defined in this way has only one descendant (see the proof of Lemma 7.8.7), any @ node has a left and a right descendant, and any λ-node has one descendant. We left to find the correspondence between variables and binders. For each λ-node $\chi\lambda$, there is a v-node u with an input edge for any $\phi e \lambda$ that $\chi\lambda$ splits into $\chi\lambda\psi e\lambda$, in accord with the last proviso in Definition 7.8.6. The unshared graph obtained in this way is proper, according to Definition 7.8.4, and is a complete unsharing of G.

The previous procedure to get a complete unsharing of G from its proper paths might seem non-deterministic, for we could have more than one occurrence of the same node λ splitting $\phi e \lambda$. The next lemma proves that this is not the case (we invite the reader to compare this lemma with the nesting property used proving correctness of Lamping's algorithm in section 3.5.1).

Lemma 7.8.8 *Let G be a proper sharing graph and let $\phi u e v, \psi u e v \in \Pi(G)$ (i.e., two proper paths ending with the same edge). Let $\mathcal{F}_{\phi u e v}$ and $\mathcal{F}_{\psi u e v}$ be the corresponding transfer functions (the internal state of G is not relevant and might even be different for the two cases). If $\mathcal{F}_{\phi v u e v} = \mathcal{F}_{\psi u e v}$, then $\phi = \psi$.*

Proof The proof is by induction on the length of ϕ, ψ and by case analysis of the type of u. The base case is trivial: e is the edge at the root of G and there is nothing to prove. Therefore, let $\phi = \chi' u' e' u$ and $\psi = \chi'' u'' e'' u$. The statement follows by induction hypothesis once showed that $e' = e''$ and $\mathcal{F}_\phi = \mathcal{F}_\psi$. The only relevant cases are:

- u *is a positive mux.* The edge e is the principal edge of the mux u, while e' and e'' are connected to two auxiliary ports, say that (a', q') and (a'', q'') are the corresponding name-offset pairs. Let m be the threshold of u. By hypothesis, $\mathcal{F}_{\phi u e v} = \mathcal{L}[m, q', a']\,\mathcal{F}_\phi = \mathcal{H} = \mathcal{L}[m, q'', a'']\,\mathcal{F}_\psi = \mathcal{F}_{\psi u e v}$; that, since $\mathcal{H} \neq 0$, is possible only if $a' = a''$ and $q' = q''$, that is, $e' = e''$. Then by simplification of the lifting operator, we also get $\mathcal{F}_\psi = \mathcal{F}_\phi$.
- u *is a negative mux.* The edge e is in this case connected to an auxiliary port of u, say that (a, q) is the corresponding name-offset pair; the edges e' and e'' are instead the principal edge of u, that is, $e' = e''$. By hypothesis, $\mathcal{F}_{\phi u e v} = \overline{\mathcal{L}}[m, q, a]\,\mathcal{F}_\phi = \mathcal{H} = \overline{\mathcal{L}}[m, q, a]\,\mathcal{F}_\psi = \mathcal{F}_{\psi u e v}$; that, since $\mathcal{H} \neq 0$, is possible only if $\mathcal{F}_\phi = \mathcal{L}[m, q, a]\,\mathcal{H} = \mathcal{F}_\psi$.
- e *is the right edge of the* @ *node* u. Also in this case we immediately have $e' = e''$ (it is the context edge of u). By hypothesis of properness, $\mathcal{F}_{\chi'}, \mathcal{F}_{\chi''} \in \mathsf{LSeq}[0, n]$ where n is the level of u, and $\mathcal{S}'\mathcal{F}_\phi = \mathcal{S}''\mathcal{F}_\psi$ for some $\mathcal{S}', \mathcal{S}'' \in \mathsf{LSeq}[n, n+1]$. But, by Lemma 7.8.3, this is possible if only if $\mathcal{F}_\phi = \mathcal{F}_\psi$ and $\mathcal{S}' = \mathcal{S}''$.

The other cases are trivial. □

By inspection of the previous proof, we see that not only ϕ and ψ coincide, but also the internal states for which $\mathcal{F}_{\phi v u e v}$ and $\mathcal{F}_{\psi u e v}$ were computed are equal, at least for the part relative to ϕ and ψ. Actually, for the proof of the next theorem it would have been enough to prove Lemma 7.8.8 for a particular internal state, say 1; that on the other hand, is the relevant one for the determination of the unshared instance corresponding to a sharing graph (see Exercise 7.8.16). The other states are rather relevant for the proof of invariance of properness under β-reduction.

Theorem 7.8.9 *Let* G *be a proper sharing graph. Its complete unsharing is unique.*

Proof The only thing that must be proved is that there is a unique way to split any proper path ϕ from the root to a variable into the

concatenation of an access path χ from the root to the node λ binding the variable and a loop ψ from the body to the binding port of λ.
Let $\phi \in \Pi(G)$ be a maximal path. If λ (let n be its index) is the node to which the ν-node is back-connected, let us assume that for both $\phi = \chi\lambda\psi$ and $\phi = \chi'\lambda\psi'$ the third proviso of Definition 7.8.6 holds. Fixed an internal state (say 1), if q is the offset of the variable reached by ϕ, we have that $\mathcal{S}\mathcal{F}_\chi = \mathcal{F}_\phi = \mathcal{S}'\mathcal{F}_{\chi'}$, for some $\mathcal{S}, \mathcal{S}' \in \mathsf{LSeq}[n, n+q+1]$, with $\mathcal{F}_\chi, \mathcal{F}_{\chi'} \in \mathsf{LSeq}[0, n]$. But, by Lemma 7.8.3, this is possible only if $\mathcal{F}_\chi = \mathcal{F}_{\chi'}$. Therefore, by Lemma 7.8.8, we conclude $\chi = \chi'$ and $\psi = \psi'$. □

We invite the reader to compare the previous proof with the analog one in Lamping's case.

Exercise 7.8.10 Give an algorithm that verify if a sharing graph is proper and compute its complete unsharing.

7.8.1 Deadlock-freeness and properness

In the following properness refers to the semantical property of sharing graphs given by the two equivalent characterizations in Definition 7.8.4 and Definition 7.8.6.

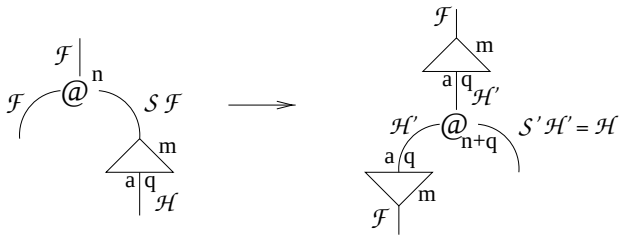
Properness is mainly an algebraic characterization of deadlock-freeness. But, while the definition of deadlock-freeness does not give any hint on how to verify it, the equivalence with properness that we are going to prove will show that deadlock-freeness is decidable, for we already gave an algorithm to verify properness.

The easy thing to prove is that properness is well-defined w.r.t. π-reduction. Let us start with the case of unshared graphs.

Lemma 7.8.11 *Properness of unshared graphs is invariant under π-reduction.*

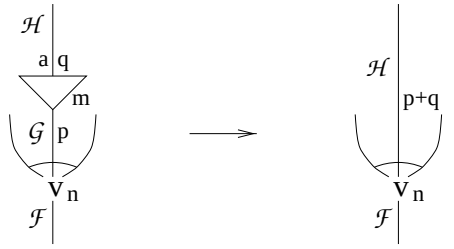
Proof By inspection of π-rules. Let us give some relevant cases only.
Let U be an unshared proper graph such that $r : U \to_\pi U'$, where r is a π-rule involving an n-indexed @-node $\nu_@$, say a negative lift ν_Δ connected to the argument port of $\nu_@$. We start proving that if U is proper, then U' is proper too.
If σ is any internal state of U, let $\mathcal{S}$ be the lifting sequence it associates to $\nu_@$ and $\mathcal{L}[m, q, a]$ be the lifting operator of ν_Δ. Let $\mathcal{F}$ and $\mathcal{H}$ be

the lifting sequences that the assignment for σ associates to the edges incident to $v_@$ and v_Δ in U, according to the labeling reported on the left-hand side of the following picture.



By the side condition of the rule, we know that $m < i$; while, by the definition of lifting assignment, we see that $\mathcal{S}\mathcal{F} = \mathcal{L}[m, q, a]\,\mathcal{H}$, with $\mathcal{F} \in \mathsf{LSeq}[0, n]$ and $\mathcal{S} \in \mathsf{LSeq}[n, n+1]$.
According to the latter considerations, we also have that $\overline{\mathcal{L}}[m, q, a]\,\mathcal{S} = \mathcal{S}'\,\overline{\mathcal{L}}[m, q, a]$, where $\mathcal{S}' = \mathcal{S}^{+q}$ is the lifting sequence obtained from $\mathcal{S}$ increasing by q all the thresholds in $\mathcal{S}$. Moreover, it is immediate to see that $\mathcal{S}' \in \mathsf{LSeq}[n+q, n+q+1]$.
Let us note that $\mathcal{H} = \overline{\mathcal{L}}[m, q, a]\,\mathcal{S}\mathcal{F} = \mathcal{S}'\,\overline{\mathcal{L}}[m, q, a]\,\mathcal{F}$; that, by the AB^* property of lifting sequences, is possible only if $\mathcal{F} = \mathcal{L}[m, q, a]\,\mathcal{H}'$, for some $\mathcal{H}' \in \mathsf{LSeq}[0, n+q]$ (we leave to the reader the easy verification of the fact that $\mathcal{H}' \in \mathsf{LSeq}[0, n+q]$).
Summarizing, the lifting sequences $\mathcal{H} = \mathcal{S}'\mathcal{H}'$, $\mathcal{H}'$ and $\mathcal{F}'$ can be assigned to the edges incident to the residuals of $v_@$ and v_Δ in U' according to the labeling reported in the right-hand side of the picture above. This assignment satisfies the constraints of the nodes inserted in U' by the reduction of the redex r and has not influence on the lifting sequences that the assignment for σ associated in U to the other edges. We get in this way a lifting assignment of U' for the internal state σ' of U' obtained from σ by replacing $\mathcal{S}'$ for $\mathcal{S}$.
It is indeed immediate to check that also the converse is trivially true. Namely, if the assignment of U' for the state σ' associates to the edges inserted by the reduction of r the lifting sequences in the right-hand side of the above picture, then the lifting sequences in the left-hand side give an assignment of U for the internal state σ.
To conclude the invariance of properness under the previous rule, we left to note that any $\mathcal{S}' \in \mathsf{LSeq}[n+q, n+q+1]$ can be obtained increasing by q the thresholds of the lifting operator in some $\mathcal{S} \in \mathsf{LSeq}[n, n+1]$, and vice versa.

Let us now consider the case in which $\mathsf{U} \rightarrow_\pi \mathsf{U}'$ by an absorption rule. The situation is as drawn in the following picture:

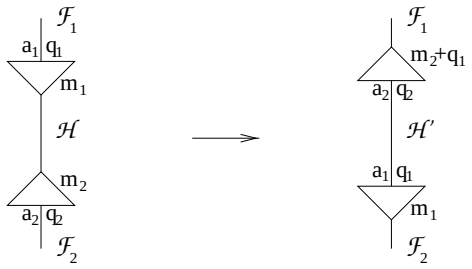


in which $n \leq m < n + p + 1$.

Let $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$ be the lifting sequences that an assignment of U associates to the edges in the left-hand-side of the above picture. By definition of lifting assignment, we have that $\mathcal{G} = \mathcal{S}\,\mathcal{F} = \mathcal{L}[m, q, a]\,\mathcal{H}$, where $\mathcal{F} \in \mathsf{LSeq}[0, n]$ and $\mathcal{S} \in \mathsf{LSeq}[n, n + p + 1]$. In particular, since $n \leq m$, an easy calculation allow to conclude that necessarily $\mathcal{S} = \mathcal{L}[m, q, a]\,\mathcal{S}'$, with $\mathcal{S}' \in \mathsf{LSeq}[n, n+p+q+1]$, and then that $\mathcal{H} = \mathcal{S}'\,\mathcal{F}$. Therefore, the lifting sequences associated to the right-hand-side in the above picture give a lifting assignment of U' (the lifting sequences of the other edges are unchanged, internal state of @-nodes included). Hence, for any assignment of U for an internal state σ we get an assignment of U' for the same internal state.

The converse is also immediate. In this case, the assignment for U' gives $\mathcal{F}$ and $\mathcal{H}$; thus, to obtain an assignment for U, it suffices to take $\mathcal{G} = \mathcal{L}[m, q, a]\,\mathcal{H}$.

The other cases are similar, therefore, as a final example we let us analyze the swap rule of lifts only. The situation is as in the following picture:



in which $m_1 < m_2$. The left-hand-side gives $\mathcal{H} = \mathcal{L}[m_1, q_1, a_1]\,\mathcal{F}_1$; the right-hand-side gives instead $\mathcal{L}[m_2 + q_1, q_2, a_2]\,\mathcal{H}' = \mathcal{F}_1$; by a simple replacement of $\mathcal{F}_1$ we get then $\mathcal{H} = \mathcal{L}[m_1, q_1, a_1]\,\mathcal{L}[m_2 + q_1, q_2, a_2]\,\mathcal{H}'$.

Therefore,, given a lifting assignment of U', and in particular $\mathcal{H}'$, the previous equation ensures the existence of an $\mathcal{H}$ satisfying the constraints of the lifts in U (in particular, note that $\mathcal{H} = \mathcal{L}[\mathfrak{m}_2, \mathfrak{q}_2, \mathfrak{a}_2]\,\mathcal{F}_2$, for $\mathcal{L}[\mathfrak{m}_1, \mathfrak{q}_1, \mathfrak{a}_1]\,\mathcal{L}[\mathfrak{m}_2 + \mathfrak{q}_1, \mathfrak{q}_2, \mathfrak{a}_2] = \mathcal{L}[\mathfrak{m}_2, \mathfrak{q}_2, \mathfrak{a}_2]\,\mathcal{L}[\mathfrak{m}_1, \mathfrak{q}_1, \mathfrak{a}_1]$). Vice versa, the AB^* property ensures that, given a lifting assignment of U', and in particular given $\mathcal{H}$, there exists $\mathcal{H}'$ for which the constraints of the lifts in U' are satisfied.
The proof of the other cases is similar and is thus omitted. □

An immediate consequence of the previous lemma is that π-reduction of unshared graphs never get stuck.

Lemma 7.8.12 *Any proper unshared graph is deadlock-free.*

Proof Let U be a proper unshared graph. The constraints imposed by the existence of a lifting assignment for any internal state of U (actually, the state $\mathfrak{1}$ would suffice) are incompatible with the presence of deadlocked redexes in U. Hence, U does not contain any deadlock. By Lemma 7.8.11, if $\mathsf{U} \twoheadrightarrow_\pi \mathsf{U}'$, also U' is proper and, because of the previous remark, does not contain deadlocks. □

The simulation property for π (Lemma 7.4.3) allows to extend the previous results to sharing graphs too and to conclude that properness implies deadlock-freeness.

Proposition 7.8.13 *Properness of sharing graphs is invariant under π-reduction and any proper sharing graph is deadlock-free.*

Proof Let G be a proper sharing graph. We already remarked (see the end of section 7.4) that, when G contains a deadlock, each G' s.t. $\mathsf{G}' \preccurlyeq \mathsf{G}$ contains at least a deadlocked redex. Therefore, since the proper unshared graph U s.t. $\mathsf{U} \preccurlyeq \mathsf{G}$ is deadlock-free (by Lemma 7.8.12), we conclude that G' does not contain deadlocks too.
By the simulation lemma for π (Lemma 7.4.3), whenever $\mathsf{G} \twoheadrightarrow_\pi \mathsf{G}'$ there is a corresponding reduction $\mathsf{U} \twoheadrightarrow_\pi \mathsf{U}'$ such that $\mathsf{U}' \preccurlyeq \mathsf{G}'$. By the invariance of properness for unshared graphs, U' is proper and thus, so is G'; moreover, the fact that U' is deadlock-free implies that G' cannot contain deadlocks. □

To conclude the analysis of π-rules, we left to prove that they are strongly normalizing on proper sharing graphs.

Proposition 7.8.14 *The π-rules are strongly normalizing over proper sharing graphs.*

Proof Because of Proposition 7.4.4, it suffices to prove the result for the unshared case.
Let U be a proper unshared graph. The proof rests on the idea that each lifting sequence of an assignment corresponds to a product of operators that must cross and reindex the corresponding edge along the reduction. According to this, we expect that each π-rule decrease the length of at least one of that lifting sequences or, in the case of absorption and annihilation, something taking into account the number of lifts in the graph. The previous considerations formalize in the following two measures, defined for the assignment of U corresponding to the internal state 1: (a) k_1 is the sum of the lengths of the lifting sequences assigned to context edges of nodes λ or @, or to principal edges of v-nodes; (b) k_2 is the sum of the lengths of the lifting sequences assigned to the principal edges of the lifts in the graph.
By inspection of the rules, we see that, with the exception of absorption, each π-rule involving a proper node decreases k_1, though it generally increases k_2. The other π-rules decrease instead k_2 leaving k_1 unchanged. For instance, let us take the two examples described in the proving Lemma 7.8.11.
In the rule for the @-node, the lifting sequence its context edge in U is $\mathcal{F}$, while in U' is $\mathcal{H}'$. All the other lifting sequences of context edges are unchanged, and moreover, since $\mathcal{F} = \mathcal{L}[m, q, a]\,\mathcal{H}'$, $|\mathcal{F}| > |\mathcal{H}'|$. Therefore, the rule decrease k_1. At the same time, since the lift has two residuals, the addend $|\mathcal{F}|$ in k_2 (remind that $\mathcal{S} = 1$) is replaced by $2|\mathcal{F}|$, causing an increase of k_2.
In the absorption rule, k_1 is obviously unchanged, while k_2 decreases for the reduction erases one lift, causing the corresponding erasing of the addend $|\mathcal{G}|$ from k_2 (note also that $|\mathcal{G}| > 0$).
Finally, also the swap rule of lifts does not change k_1, while two addend $|\mathcal{H}|$ in k_2 are replaced by $|\mathcal{F}_1|$ and $|\mathcal{F}_2|$ with $\mathcal{H} = \mathcal{L}[m_i, q_i, a_i]\,\mathcal{F}_i$, for $i = 1, 2$.
Summarizing, each π rule decreases the combined measure (k_1, k_2) (w.r.t. lexicographic ordering). Therefore, since $(k_1, k_2) \geq (0, 0)$, there is no infinite π-reduction of any proper unshared graph. □

The previous propositions and lemmas sum up in the following theorem, that states indeed the equivalence between the semantical (Defini-

tion 7.8.4 and Definition 7.8.6) and syntactical (Definition 7.5.1) characterizations of properness.

Proposition 7.8.15 *Any proper sharing graph* G *has a unique* π *normal-form* $\mathcal{R}(\mathsf{G}) = [\![\mathsf{U}]\!]$, *where* U *is the complete unsharing of* G.

Proof Since any proper sharing graph G has an unshared instance $\mathsf{U} \preccurlyeq \mathsf{G}$ that is deadlock-free, by Proposition 7.4.4 we conclude that the normal form $\mathcal{R}(\mathsf{G})$ of G there exists and is unique; moreover, $\mathcal{R}(\mathsf{G})$ does not contain lifts or muxes. This last fact, in addition with the simulation property for π (Lemma 7.4.3, also implies that $\mathcal{R}(\mathsf{G}) = \mathcal{R}(\mathsf{U})$.
By definition of $[\![\,.\,]\!]$ (see Exercise 7.5.2), we see that the λ-tree $[\![\mathsf{U}]\!]$ is invariant under π-reduction (i.e., $[\![\mathsf{U}']\!] = [\![\mathsf{U}]\!]$, for any pair of unshared graph such that $\mathsf{U} \twoheadrightarrow_\pi \mathsf{U}'$). In particular, since $\mathcal{R}(\mathsf{G})$ is a λ-tree, $[\![\mathcal{R}(\mathsf{G})]\!] = \mathcal{R}(\mathsf{G})$. By the fact that $\mathsf{U} \twoheadrightarrow_\pi \mathcal{R}(\mathsf{G})$, we eventually get $\mathcal{R}(\mathsf{G}) = [\![\mathsf{U}]\!]$.
To conclude, let us note that the invariance of $[\![\mathsf{U}]\!]$ gives indeed a direct proof of the uniqueness of the π normal form. □

Exercise 7.8.16 Let $[\![\,.\,]\!]^\bullet$ be the map that associates a λ-tree $[\![\mathsf{U}]\!]$ to any proper unshared graph U in the following way:

(i) Replace all the lifts in U by direct connections between their ports.

(ii) For any edge e connected to the context port of a λ or @ node or to any port of a v-node, denote by $\mathcal{F}_\mathsf{e}$ the transfer function for the internal state 1 of the (unique) proper path ending with e, and by $\|\mathcal{H}_\mathsf{e}\|$ the sum of all the offsets in $\mathcal{H}_\mathsf{e}$. Reindex nodes and v-node offsets according to the following rules:

  (a) The new index of any n-indexed @ or λ-node is equal to $\mathsf{n} + \|\mathcal{H}_\mathsf{e}\|$, where e is the context edge of the node. The new index of any v-node is equal $\mathsf{n} + \|\mathcal{H}_\mathsf{e}\|$, where e is the principal edge of the node.

  (b) The new offset of an edge e entering into an n-indexed v-node is $\mathsf{q} + \|\mathcal{S}_\mathsf{e}\|$, where q is the old offset of e and $\mathcal{S}_\mathsf{e} \in \mathsf{LSeq}[\mathsf{n}, \mathsf{n}+\mathsf{q}+1]$ is the unique lifting sequence such that $\mathcal{F}_\mathsf{e} = \mathcal{S}_\mathsf{e}\mathcal{F}$, for some $\mathcal{F} \in \mathsf{LSeq}[0, \mathsf{n}]$.

Prove that:

(i) $[\![\mathsf{U}]\!]^\bullet$ is a λ-tree;

(ii) $[\![\mathsf{U}]\!]^\bullet$ is invariant under π-reduction;

(iii) $[\![\mathsf{U}]\!]^\bullet = [\![\mathsf{U}]\!] = \mathcal{R}(\mathsf{U})$.

7.9 Soundness and adequateness

The next step is to prove that properness is sound w.r.t. to β-rule too. The key point is to prove soundness of β_u.

Lemma 7.9.1 *Let* U *be a proper unshared graph. Any unshared graph* U' *s.t.* $U \to_{\beta_u} U'$ *is proper.*

Proof Let us just consider the case in which the variable involved in the rule has only one occurrence. The generalization to the k-ary case is immediate, for it suffices to repeat the argument for any of the occurrences of the variable.
Any path ϕ' starting at the root of U' has an ancestor in U (starting at the root of U). We can distinguish the following cases, according to the position of the redex u:

(i) ϕ *did not traverse any node of* u. In this case ϕ and ϕ' are isomorphic and the properness of ϕ' follows immediately from the properness of ϕ.

(ii) ϕ *traversed the edge* u. The only difference between ϕ and ϕ' is that the sequence of three edges traversing the @-λ pair of u that was in ϕ (i.e., the context edge of u, the edge u itself and the body edge of u) has been merged into a unique edge in ϕ'. It is however immediate to note that this substitution has no impact on the properness of ϕ', for the transfer function of the removed @-λ pair was the identity.

(iii) ϕ *traversed the @-node to enter the argument part of* u. Let $\xi\nu e\lambda$ be the path of U starting at the root and ending at the binding port of the λ-node λ in u. The syntax ensures that ξ crosses the λ-node λ, that is, $\xi = \chi\lambda\psi\nu$. Moreover, by the properness of U, for any state σ, there exists $\mathcal{S}[\sigma] \in \mathsf{LSeq}[n, n+q+1]$ s.t. $\mathcal{F}_\xi[\sigma] = \mathcal{S}[\sigma]\mathcal{F}_\chi[\sigma]$, where n is the level of λ and q the offset of the port reached by ξ.
Let $\phi = \chi@\zeta$ be the crossing of the @-node @ under analysis. The path ϕ' is obtained by replacing the path $\psi' e_\Delta \nu_\Delta$ for the node @, where ψ' is ψ without its first edge (the body edge of u), ν_Δ is a lift with threshold n and offset q, and the edge e_Δ is connected to the auxiliary port of ν_Δ (let a be the name of this port). According to this, ϕ' splits in the following way $\phi' = \chi' \cdot e \cdot \psi' e_\Delta \nu_\Delta \zeta'$, where χ' is isomorphic to χ without its last edge (the context edge of u), ζ' is isomorphic to ζ without its first edge (the argument

edge of u), and e is the edge inserted by the reduction to replace the @-λ pair of u.
Let us now compute the transfer functions of the previous paths. The path $\xi' = \chi' \cdot e_1 \cdot \psi'$ is the image in U' of the path $\xi = \chi\lambda\psi$ of U. Then, by the previous item in the proof, we also know that its transfer function is $\mathcal{F}_{\xi'}[\sigma] = \mathcal{F}_{\xi}[\sigma] = \mathcal{S}[\sigma]\mathcal{F}_{\chi}[\sigma]$, for some $\mathcal{S}[\sigma] \in \mathsf{LSeq}[n, n+q+1]$. The transfer function of ξ' followed by the lift inserted by the reduction is then $\mathcal{F}_{\xi' e_\Delta}[\sigma] = \mathcal{L}[n, q, a]\mathcal{S}[\sigma]\mathcal{F}_{\chi}[\sigma]$, and an easy calculation allows to verify that $\mathcal{L}[n, q, a]\mathcal{S}[\sigma] \in \mathsf{LSeq}[n, n+1]$. Therefore, assuming that σ associate the internal state $\mathcal{L}[n, q, a]\mathcal{S}[\sigma]$ to the @-node @ in the redex u, we see that $\mathcal{F}_{\phi}[\sigma] = \mathcal{F}_{\phi'}[\sigma]$ (let us remark that here, and in all the previous equations, we have implicitly assumed that the internal state σ of U' is the natural one obtained from the state σ of U erasing the value for @). Since apart for the value assigned to @ the state σ is arbitrary, we conclude the properness of U'.

□

The extension of the previous result to sharing graphs is immediate because of the simulation property for β. As a matter of fact, since we already shown that properness is sound for π-rules too, we can directly state the result for the whole system.

Proposition 7.9.2 *Let* G *be a proper sharing graph. If* $G \twoheadrightarrow G'$*, then* G' *is proper.*

Proof It is an immediate consequence of the simulation properties (Lemma 7.4.3 and Lemma 7.5.4) and of the invariance of properness for unshared graphs under β_u-reduction (Lemma 7.9.1) and π-rules (Lemma 7.8.11). □

We can eventually prove that sharing reduction is sound w.r.t. standard λ-calculus β-reduction and that, in the extended (and thus non-optimal) system, any λ-calculus reduction can be implemented on sharing graphs.

Theorem 7.9.3 *Let* T *be a* λ*-tree.*

(i) *For any sharing reduction* $T \twoheadrightarrow G$*, there exists a standard* λ*-calculus reduction s.t.* $T \twoheadrightarrow_\lambda \mathcal{R}(G)$.

(ii) *For any* λ*-calculus reduction* $T \twoheadrightarrow_\lambda T'$*, there exists a sharing reduction s.t.* $T \twoheadrightarrow T'$

Proof First of all let us note that T, and then G, is proper (see Exercise 7.8.5) and that $\mathcal{R}(\mathsf{T}) = \mathsf{T}$.
The first item follows by the following observations:

(i) For any proper sharing graph G', if $\mathsf{G}' \to_\pi \mathsf{G}$, then $\mathcal{R}(\mathsf{G}) = \mathcal{R}(\mathsf{G})'$. By the uniqueness of the π-normal form.

(ii) For any proper sharing graph G', if $\mathsf{G}' \to_\beta \mathsf{G}$, then $\mathcal{R}(\mathsf{G}) \to_\lambda \mathcal{R}(\mathsf{G})'$. In fact, by Proposition 7.8.15, G' and G have two unique π normal forms, that are indeed the π normal forms of the respective complete unsharings U and U' (i.e., $\mathcal{R}(\mathsf{G}) = \mathcal{R}(\mathsf{U})$ and $\mathcal{R}(\mathsf{G}') = \mathcal{R}(\mathsf{U}')$, with $\mathsf{U} \preccurlyeq \mathsf{G}$ and $\mathsf{U}' \preccurlyeq \mathsf{G}$); moreover, by Exercise 7.5.2 and simulation property of β, $\mathcal{R}(\mathsf{G}) \twoheadrightarrow_\lambda \mathcal{R}(\mathsf{G}')$.

Then an easy induction on the length of the reduction of T allows to conclude that $\mathsf{T} \twoheadrightarrow_\lambda \mathcal{R}(\mathsf{G})$.
The second item is instead an immediate consequence of Exercise 7.3.3. □

7.10 Read-back and optimality

The relevant point of the results proved in the previous sections is the uniqueness of the π-normal form of any proper sharing graph G and the fact that it coincides with the λ-term matching such a graph. Then, since every sharing graph obtained along the reduction of a λ-tree is proper, the π-rules give an implementation of the read-back algorithm. This unrestricted application of the π-rules causes however the loss of all the sharing accumulated in G—as a matter of fact, it could not be otherwise, for we want to get a λ-tree as the result of read-back.

In spite of this, optimality can be recovered assuming to follow a lazy reduction strategy in the application of π-rules. The idea is to execute a mux propagation only when it might hide a β-redex. Namely, given a sharing graph G such that $\mathcal{R}(\mathsf{G}) = \mathsf{T}$, a propagation rule should be applied on G only if the corresponding muxes might hide a β-rule present in T.

For instance, the two binary muxes in Figure 7.28 hide in G at least a pair of redexes contained in the corresponding λ-term T. In more detail, if the two muxes are complementary, after an annihilation we immediately get a pair of β-redexes; while, if the two muxes are not complementary (we assume anyhow that they are not deadlocked), after a swap and two propagations we get four β-redexes. Since all the

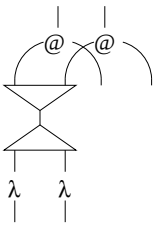


Fig. 7.28. Hidden β-rules.

previous rules does not change the read-back, that redexes were already explicit in T.

According to this lazy approach, it is an easy observation to notice that the rules in Figure 7.29 are the minimal subset ensuring that situations as the one previously described never get stuck, inadvertently ending the computation when it should rather proceed—note that the optimal rules contain the propagation through the principal node of a v-node in order to ensure to completely explicit β-redexes.

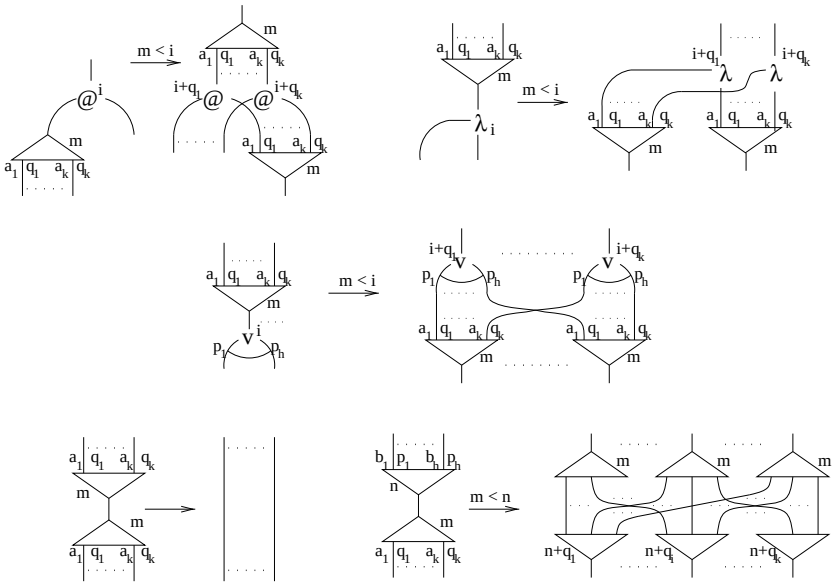


Fig. 7.29. Optimal π-rules.

It should not be surprising that the previous system corresponds ex-

actly to the rules that we gave in Chapter 3 reformulated according to the notation based on muxes. Moreover, it is also immediate to note that the addition of absorption to the previous set does not cause the loss of optimality. Indeed, this is a first example of a more general technique called *safe operators* that we will study in Chapter 9.

As the standard Lamping's algorithm, the optimal π-rules in Figure 7.29 are an interaction net. Therefore, they are trivially locally confluent. The addition of absorption improves efficiency, but from the theoretical point of view has the side-effect of the loss of confluence: it is an easy exercise to build an example in which the optimal rules plus absorption are not confluent, starting from a critical pair between an absorption and a β-rule.

The whole system of the π-rules plus β is instead confluent. In fact, given two reductions $M \twoheadrightarrow G'$ and $M \twoheadrightarrow G''$ we can always apply the read-back algorithm to obtain $\mathcal{R}(G')$ and and $\mathcal{R}(G'')$. Then, by the confluence of λ-calculus, we know that there exists a λ-tree N such that $M \twoheadrightarrow_\lambda \mathcal{R}(G') \twoheadrightarrow_\lambda N$ and $M \twoheadrightarrow_\lambda \mathcal{R}(G'') \twoheadrightarrow_\lambda N$. By Theorem 7.9.3, we finally see that the previous commuting diamond gives indeed a commuting diamond of sharing reductions composed of $M \twoheadrightarrow G' \twoheadrightarrow_\pi \mathcal{R}(G') \twoheadrightarrow_s N$ and $M \twoheadrightarrow G'' \twoheadrightarrow_\pi \mathcal{R}(G'') \twoheadrightarrow_s N$.

To conclude the chapter, let us recollect in a theorem the read-back properties of π-rules.

Theorem 7.10.1 (read-back) *Let $T \twoheadrightarrow G$ be any sharing graph reduction, e.g., an optimal one obtained by applying rules contained in the set in Figure 7.29 only.*

(i) *The π-rules are strongly normalizing and confluent on G.*

(ii) *The unique normal-form of G, say the* read-back *of G, is a λ-tree $\mathcal{R}(G)$ such that $T \twoheadrightarrow_\lambda \mathcal{R}(G)$.*

Proof Then first item is an immediate consequence of Proposition 7.8.15 and Proposition 7.9.2. The second item is just a restatement of Theorem 7.9.3. □

8

Other translations in Sharing Graphs

The encoding of λ-terms into sharing graphs which have been presented in this book *is not* the original one proposed by Lamping. There are several important reasons for departing from Lamping's approach. First of all, his algorithm contained a lot of "optimization" rules which hide the core of the issue, preventing any sensible theoretical foundation of this implementation technique†. Secondly, our presentation is aimed to stress the relation between optimal reduction and Linear Logic that is one of the most intriguing and suggestive aspects of this implementation of the λ-calculus.

Actually, since Lamping's seminal work [Lam90] on optimal graph reduction techniques for λ-calculus, several different translations based on the same set of control operators (sharing graphs) have been proposed in literature. Lamping's approach was revisited for the first time in [GAL92a], where a restricted set of control operators and reduction rules was proved sufficient for the implementation. In the same paper, an interesting and quite odd notation was introduced (the so called "bus notation"), with the aim to further minimize the number of reduction rules. In [GAL92b] the same authors pointed out a strong analogy between optimal reductions and Linear Logic [Gir87], already hinted to in their previous paper. By their encoding of Linear Logic in sharing graphs, and some encoding of λ-calculus in Linear Logic, a third (slightly different) translation was implicitly obtained. This encoding has been used (and explicited) in [AL93a], in the much more general case of Interaction Systems. In the same paper, the authors pointed out that Lamping's control nodes could be considered as a very abstract set of operators for implementing (optimal) sharing in virtually every class of

† We shall discuss these rules, which are *essential* in view of an actual implementation, in Chapter 9, where we shall introduce the crucial notion of safe operator.

higher order rewriting systems. This suggested the name of "sharing graphs", that has been adopted henceforth.

The translation used in this book was first defined in [GAL92b]; the main point of this translation was its close relation with dynamic algebras and the Geometry of Interaction for Linear Logic, which has been fully exploited in the previous chapter.

The aim of this chapter is to clarify the relations between all this different translations of λ-calculus into sharing graphs. We shall do this, by passing through the "bus notation", that will also give us the opportunity to shed some more light on very interesting properties of this (somewhat neglected) notation.

8.1 Introduction

The somewhat puzzling number of different translations of λ-calculus in sharing graphs is actually due to a double level of indeterminacy. As we pointed out several times when discussing the relation with Linear Logic, the real problem of this implementation technique is the optimal sharing of the box of Linear Logic [GAL92b]. So, we have a double problem, here:

- how to represent a box in sharing graphs;
- where to put boxes, in the encoding of a λ-term.

Let us briefly discuss the last point, first. Obviously, this problem has very little to do with optimal reduction: the real question, here, is how do we encode a λ-term in Linear Logic. The solution adopted so far in this book is based on the well known decomposition of the intuitionistic implication $A \to B$ as $!(A) \multimap B$, imposing moreover the type-isomorphism $D \cong !(D) \multimap D$. However, this is not the only solution: for instance, another approach is based on the type isomorphism $D \cong !(D \multimap D)$.In this case, a λ-term M with free variable in $x_1, \ldots, x_n$ is translated into a proof of

$$x_1 : D, \ldots x_n : D \vdash M : D$$

A variable, is just an axiom

$$x : D \vdash x : D$$

For λ-abstraction, we first make a linear abstraction, and then we build a box around the so obtained function:

$$\dfrac{\dfrac{x_1 : D, \ldots, x_{n-1} : D, x_n : D \vdash M : D}{x_1 : D \cong !(D \multimap D), \ldots, x_{n-1} : D \cong !(D \multimap D) \vdash \lambda x_n.M : D \multimap D}}{x_1 : D \cong !(D \multimap D), \ldots, x_{n-1} : D \cong !(D \multimap D) \vdash \lambda x_n.M : D \cong !(D \multimap D)}$$

For application, we apply the left introduction rule for linear implication, and then we make a dereliction on the new assumption:

$$\dfrac{\dfrac{x_1 : D, \ldots, x_n : D \vdash N : D \qquad y_1 : D, y_2 : D, \ldots, y_m : D \vdash M : D}{z : D \multimap D, x_1 : D, \ldots, x_n : D, y_2 : D, \ldots y_m : D \vdash M[^{(z\ N)}/_{y_1}] : D}}{z : D \cong !(D \multimap D), x_1 : D, \ldots, x_n : D, y_2 : D, \ldots y_m : D \vdash M[^{(z\ N)}/_{y_1}] : D}$$

According to our representation of proof nets into sharing graphs, this would give rise to the encoding in Figure 8.1 (where, as usual, $[M] = [M]_0$).

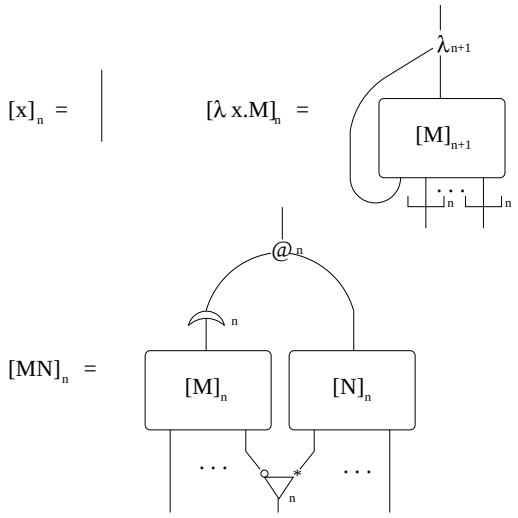


Fig. 8.1. Another encoding of λ-terms in sharing graphs

Many other encodings of λ-terms into linear logic can be defined, and each of them would provide a different initial representation into sharing graphs. The interesting fact is that all these representations would give rise to (different) optimal reductions of the λ-term, do to the optimal implementation of the underlying Linear Logic.

The more interesting aspect of the encoding is however the representation of the box in sharing graph. The solution considered so far consists in raising of one level the internal structure M of the box w.r.t. the surrounding world (see Figure 8.2(a)). A possible alternative is to use a more explicit notation, enclosing M into a set of control nodes (brackets of level 0) delimiting the extent of the box (see Figure 8.2(b)).

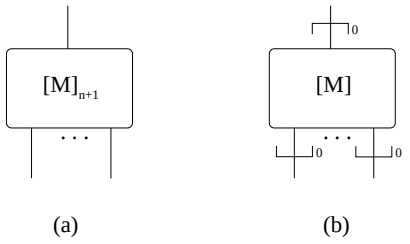


Fig. 8.2. Box representations in sharing graphs

Intuitively, this approach is somewhat dual to the former: it is not the box to be raised of one level, but the surrounding world, by the *implicit* effect of the brackets of level 0 which have been added to the graph. Note in particular that we do not need indexes any more during the translation, since both the context and the internal region of the box are (apparently) at a same level, say 0.

The control operators require now some care. For instance, the croissant, which corresponds to a dereliction, is supposed to open the box, so we should add a bracket of index 0 to annihilate the analogous operator on the input of the box; moreover, since it must propagate inside the box (which is at level 0, now), we shall translate the croissant at level 1. Similarly, a fan open the box but immediately closes it again, and a bracket open the box and create two new boxes around the term. So, according to our intuition, the three control operators should be now represented as described in Figure 8.3.

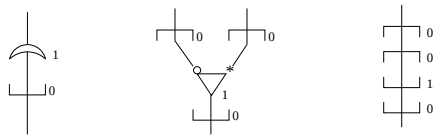


Fig. 8.3. The control operators, revisited

Putting together these ideas, and eliminating all pairs of interacting squares of level 0, we get the translation of Figure 8.4 (for closed terms).

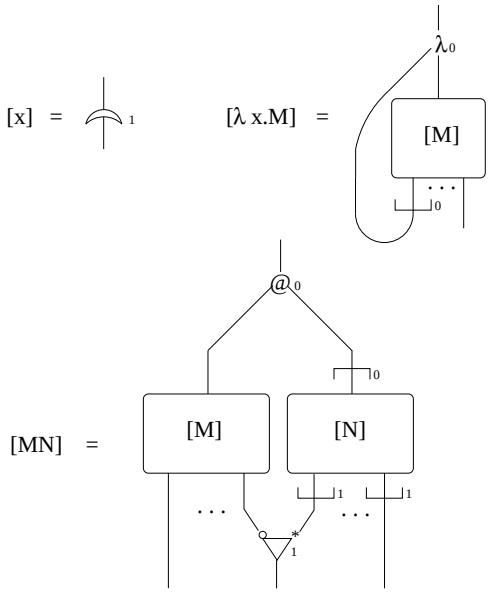


Fig. 8.4. Still another encoding of the λ-calculus

In the following, we shall denote by $\mathcal{F}(M)$ the graph generated by the translation of the term M according to Figure 8.4, and use $\mathcal{G}(M)$ for denoting our usual encoding.

It is worth noticing that the operational behavior is sensibly different in the two cases. In particular, in $\mathcal{G}(M)$, the (external) control operators which are propagating inside the box are at a lower level than the operators inside the box. The contrary happens in [GAL92b] where the square bracket of index 0 on the principal port of the box has the effect to raise the level of incoming control operators, avoiding their conflict with inner operators proper to the box. Due to this consideration, it can appear a bit surprising that *exactly the same set of rewriting rules* works in both cases. Moreover, apart the above informal discussion on the idea underlying the two box representations (i.e., their different ways to avoid conflicts), there is a priori no much evidence of a formal correspondence between them. This is exactly the issue that we shall address in the following sections. To this purpose, it is convenient to use an interesting notation, introduced for the first time in [GAL92a]: the *bus notation*.

Exercise 8.1.1 Give the translation based on the isomorphism $\mathsf{D} \cong !(\mathsf{D} \multimap \mathsf{D})$ using the box representation of $\mathcal{G}(\mathsf{M})$.

8.2 The bus notation

An alternative and very suggestive graph notation for sharing graphs is obtained by interpreting wires in the systems $\mathcal{G}(\mathsf{M})$ and $\mathcal{F}(\mathsf{M})$ as buses, namely set of wires. In this view, an $\mathfrak{i}$-indexed node is considered as an operator acting on the wire at depth $\mathfrak{i}$ (see Figure 8.5, where $_{\mathfrak{i}}\nmid$ means a bus of width $\mathfrak{i}$).

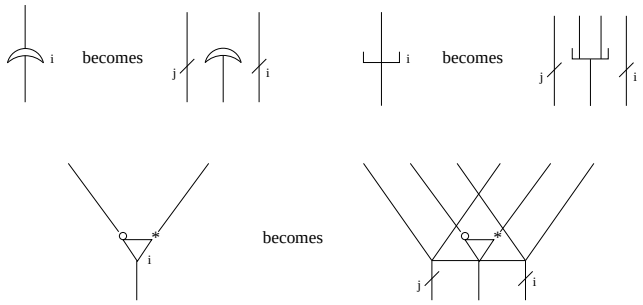


Fig. 8.5. The control operators, revisited

The rules governing the interaction of control nodes in the bus notation are depicted in Figure 8.6.

Remark that no interaction between croissants or brackets on different wires is defined, just because it is implicit in the notation. On the other side, interactions with fans are no more local, but concern the whole bus.

8.3 The bus notation of the translation $\mathcal{F}$

Trying an encoding *ad literam* through buses of the translation $\mathcal{F}$, we may immediately realize that we cannot fix a priori the dimension of the bus (we shall come back soon to this point). For the moment, after having fixed some wire "0", we shall suppose to have an infinite numbers of wires at its left. The resulting translation is defined by the function $\mathsf{B}^{\mathcal{F}}$ in Figure 8.7 (note that $\mathsf{B}^{\mathcal{F}}(\mathsf{M})$ is *exactly* the bus-counterpart of $\mathcal{F}(\mathsf{M})$).

Remark that $\mathsf{B}^{\mathcal{F}}(\mathsf{M})$ uses a *finite number* of wires. This number

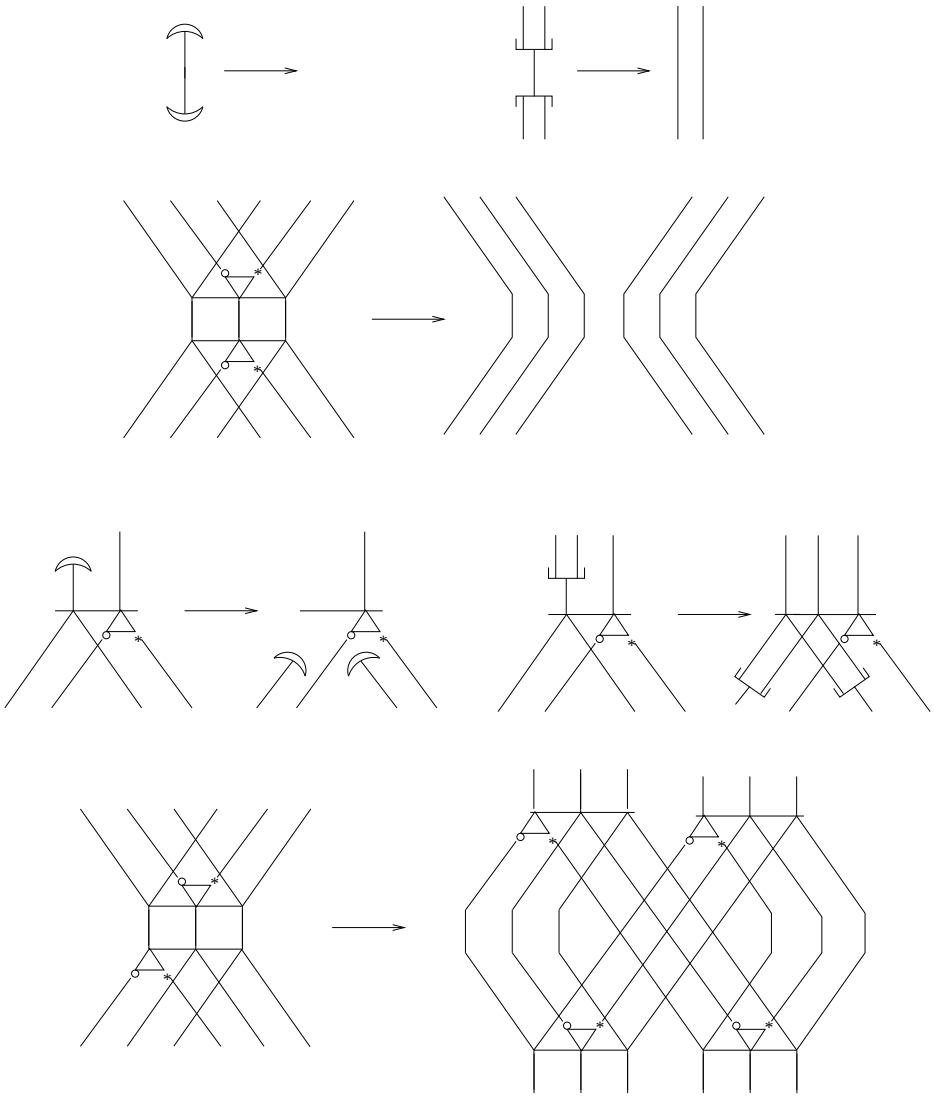


Fig. 8.6. The interactions of control nodes in bus notation

depends on the syntactical structure of the term; precisely we exactly need an extra-wire at the left every time we have a (nested) application. Alternatively we may add an index to the translation $\mathcal{F}$, mimicking the

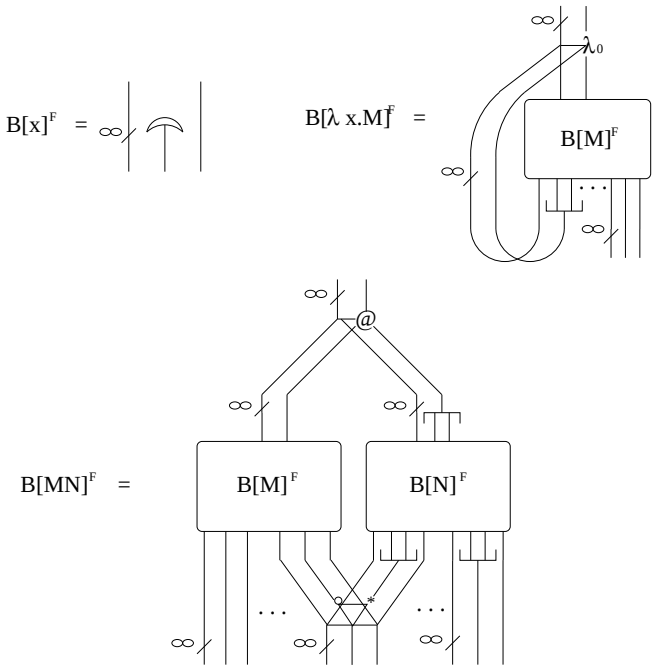


Fig. 8.7. The encoding $\mathcal{F}$ in the bus notation.

definition of $\mathcal{G}$. This index expresses now the number of extra-wires required to the left of the term (see Figure 8.9).

It is possible to avoid the infinite (or, in any case, growing) number of wires needed during the translation by introducing suitable operators (square brackets) on the leftmost wire. The resulting translation $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$ is described in Figure 8.8.

Remark that, in $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$, the bus has always dimension two at the root of a (sub-)term, and three at its free variables. The purpose of the two brackets added on the leftmost wire is exactly that of "absorbing" and "recreating" the extra-wire required by the box. This encoding was used for the first time in [GAL92a] (in conjunction with a translation of λ-terms into Linear logic based on the type isomorphism $\mathsf{D} \cong !(\mathsf{D} \multimap \mathsf{D})$).

The relation between $\mathsf{B}^{\mathcal{F}}$ and $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$ is established by Theorem 8.3.4 below. Few preliminary properties about $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$ are required.

Proposition 8.3.1 *Counting the index of operators from right to left,*

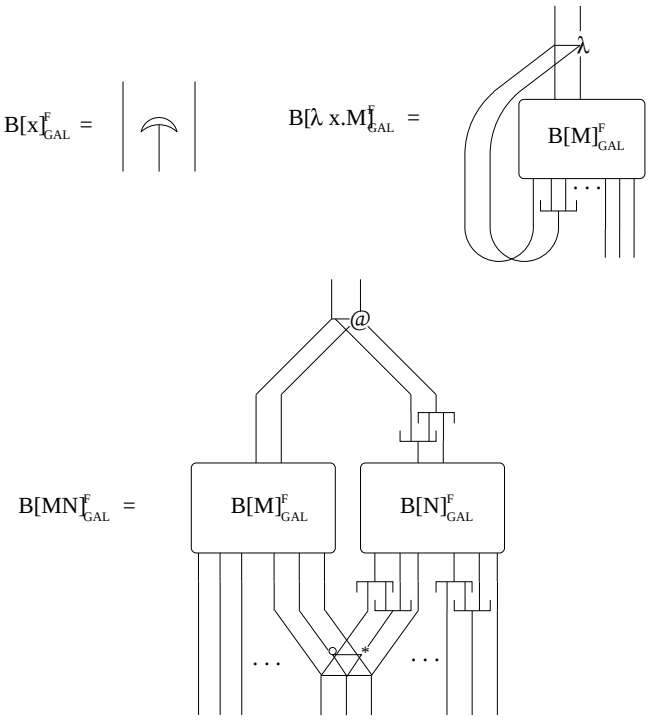


Fig. 8.8. The translation function $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$

the nodes located on the leftmost wire never modify the index of other nodes.

This is an immediate consequence of the shape of the rules in Figure 8.6.

Proposition 8.3.2 *In graphs resulting by the reduction of* λ*-terms translated according to* $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$*, the only operators which can possibly appear on the leftmost wire are brackets. Moreover such nodes will always remain on the leftmost wire.*

Also this property is immediate because it is true initially, and it is obviously preserved by the reduction rules in Figure 8.6.

Let us call a *deadlock* a pair of nodes, one in front of the other, that do not interact (i.e., no rule of Figure 8.6 can be applied). Remark that Proposition 8.3.2 guarantees the absence of deadlocks on the leftmost wires of graphs obtained by the reduction of $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}(\mathsf{M})$. Indeed, the only

nodes on the leftmost wire are brackets and two brackets, one in front of the other, can be simplified by the second rule of Figure 8.6.

Lemma 8.3.3 *There exists a correspondence between the evaluations of* $\mathsf{B}^{\mathcal{F}}(\mathsf{M})$ *and of* $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}(\mathsf{M})$. *Such correspondence just forgets the nodes on the leftmost wires of* $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$ *and the reductions concerning them.*

Indeed there exists an obvious embedding from $\mathsf{B}^{\mathcal{F}}(\mathsf{M})$ to $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}(\mathsf{M})$, which is the identity for every wire except the leftmost one. This embedding is preserved by rewritings. In particular a rewriting not involving the leftmost wire may be performed both in $\mathsf{B}^{\mathcal{F}}$ and $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$. A rewriting, let us say r, involving the leftmost wire concerns only $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$. But r does not modify the embedding for the following reasons:

(i) r does not change the level of nodes located on wires different from the leftmost (by Proposition 8.3.1);
(ii) r never duplicates nodes on wires different from the leftmost (since, by Proposition 8.3.2, a fan cannot appear on the leftmost wire).

Recall now that, in sharing graphs, brackets and croissants are used only with the purpose to guarantee the correct matching of fans and abstractions/applications. Lemma 8.3.3 implies that such matchings are performed in the same way both in $\mathsf{B}^{\mathcal{F}}$ and $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$, therefore the interactions of fans and applications/abstractions is completely insensible to the removal of leftmost brackets.

The previous remark can be formalized by proving that the semantics of $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$ is not affected by the removal of brackets on the leftmost wire, i.e., that consistent paths are insensible to this operation. We leave the proof of this fact as an easy exercise for the reader.

In conjunction with Lemma 8.3.3, we immediately get the following result:

Theorem 8.3.4 *The removal of nodes on the leftmost wires of* $\mathsf{B}^{\mathcal{F}}_{\mathrm{GAL}}$ *has no sensible effect on the evaluation of sharing graphs and on their semantics.*

8.4 The correspondence of $\mathcal{F}$ and $\mathcal{G}$

The correspondence between $\mathcal{F}$ and $\mathcal{G}$ will be proved by reasoning again on the bus-notation. The main idea underlying the proof is: *instead of*

counting levels from right to left, let us count them from left to right. The application of this criterion to the function $\mathsf{B}^{\mathcal{F}}$ gives $\mathsf{B}^{\mathcal{F},\mathrm{r}}$, which is described in Figure 8.9 (we assume $\mathsf{B}^{\mathcal{F},\mathrm{r}}(M) = \mathsf{B}_0^{\mathcal{F},\mathrm{r}}(M)$).

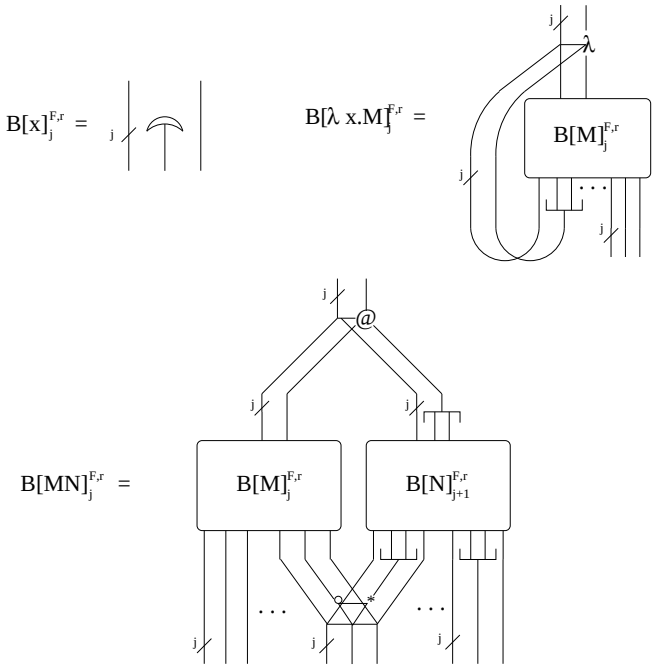


Fig. 8.9. The translation $\mathsf{B}^{\mathcal{F},\mathrm{r}}$

On the other hand, the bus interpretation $\mathsf{B}^{\mathcal{G}}$ of $\mathcal{G}$ (counting levels from left to right), is illustrated in Figure 8.10 (as usual $\mathsf{B}^{\mathcal{G}}(M) = \mathsf{B}_0^{\mathcal{G}}(M)$).

Now, $\mathsf{B}^{\mathcal{F},\mathrm{r}}$ only differs from $\mathsf{B}^{\mathcal{G}}$ for the presence of some extra-brackets at the root of the argument of an application and on bound variables. The relation between $\mathsf{B}^{\mathcal{F},\mathrm{r}}$ and $\mathsf{B}^{\mathcal{G}}$ may be fixed in the same way used in the previous section.

Lemma 8.4.1 *There exists a correspondence between the evaluations of* $\mathsf{B}^{\mathcal{G}}(M)$ *and of* $\mathsf{B}^{\mathcal{F},\mathrm{r}}(M)$. *Such correspondence just forgets the brackets on the rightmost wires of* $\mathsf{B}^{\mathcal{F},\mathrm{r}}$ *and the reductions concerning them.*

Establishing the above correspondence is a bit less obvious than for Lemma 8.3.3, because of the presence of applications and abstractions

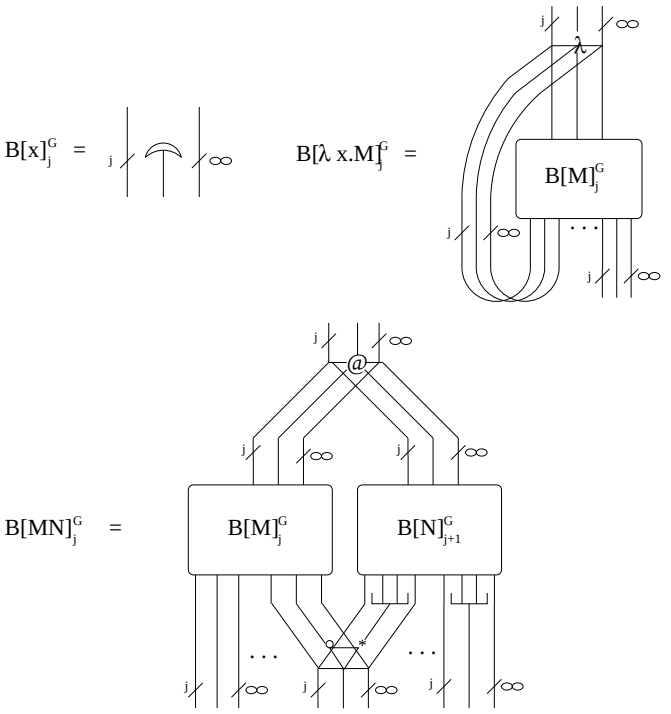


Fig. 8.10. The bus interpretation of the translation $\mathcal{G}$

on rightmost wires. In particular, what could invalidate the lemma is the eventual presence of a deadlock between a bracket and an abstraction (or application) node on the rightmost wire. Note however that there exists a bijective correspondence between abstraction/application nodes and brackets on rightmost wires. When an abstraction node, let us say $\mathtt{m}$, is fired with an application, call it $\mathtt{n}$, the bracket corresponding to $\mathtt{m}$ eventually annihilates against the bracket corresponding to $\mathtt{n}$ (i.e., they interact through the second rule of Figure 8.6). Therefore the bijective correspondence is preserved by reduction. So, we can conclude the following proposition:

Proposition 8.4.2 *If a graph is obtained by the reduction of* $\mathsf{B}^{\mathcal{F},\mathrm{r}}(\mathsf{M})$ *for some* λ*-term* M*, we can never have a deadlock between abstractions (applications) and brackets.*

Again, it is easy to prove that consistent paths are insensible to the

removal of brackets on the rightmost wires. This fact and Lemma 8.4.1 imply:

Theorem 8.4.3 *The removal of the square brackets on the rightmost wires of* $\mathsf{B}^{\mathcal{F},\mathrm{r}}$ *has no sensible effect on the evaluation of sharing graphs and on their semantics.*

8.5 Concluding remarks

A lot of different encodings of λ-terms into sharing graphs are possible, and have been actually proposed in the literature. These encodings essentially differ from each other for two respects: the way λ-terms are embedded into Linear Logic, and the way used for counting levels in the bus-representation of the graph (maybe adding or deleting "absorption" nodes on rightmost and leftmost wires).

The following schema classifies the main translations proposed in the literature, according to the above criteria:

isomorphism	right-to-left	bus	left-to-right
$\mathsf{D} \cong (!\mathsf{D}) \multimap \mathsf{D}$	[AL93a]	[AL95a]	[Asp94]
$\mathsf{D} \cong !(\mathsf{D} \multimap \mathsf{D})$	[GAL92b]	[GAL92a]	

Fig. 8.11. Classification of translations

9
Safe Nodes

In this chapter, we shall discuss the complex problem of accumulation of control operators (square brackets and croissants). The solution we shall propose (based on the so called *safe* nodes) is not yet quite satisfactory from the theoretical point of view, although it works pretty well in practice. In particular, our "tagging" method for recognizing safe nodes can be surely improved. As a matter of fact, the accumulation of control operators (and, more generally, the definitive understanding of the "oracle" in Lamping's algorithm) is still the main open problem of this technique.

9.1 Accumulation of control operators

Let us consider the term λ-term ($\underline{2}$ I) represented in Figure 9.1 (note that ($\underline{2}$ I) → I).

Its reduction starts with the complete duplication of the identity (see Figure 9.2) and ends up with the graph in Figure 9.3(a), while one would obviously expect to arrive to the configuration in Figure 9.3(b).

Actually, these two graphs are equivalent w.r.t. the standard semantics for sharing graphs provided by means of contexts [Lam90, GAL92a, ADLR94] (intuitively, the first one contains operators that open and then temporarily close levels that are never to be opened again). This accumulation of "redundant" control operators (such as in the first graph above) is the main source of inefficiency in the optimal graph reduction technique. For instance, iterating once more the application of $\underline{2}$ to the "identity" of Figure 9.3(a), the computation ends up with the "identity" of Figure 9.4, and so on, essentially doubling the number of control operators at each new application of $\underline{2}$.

In particular, in the graph rewriting system considered so far, the

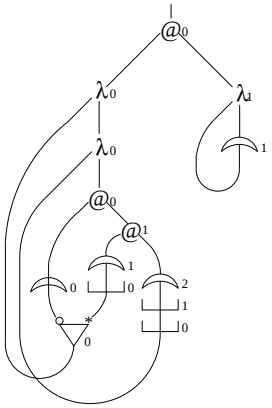


Fig. 9.1. λ-term ($\underline{2}$ I)

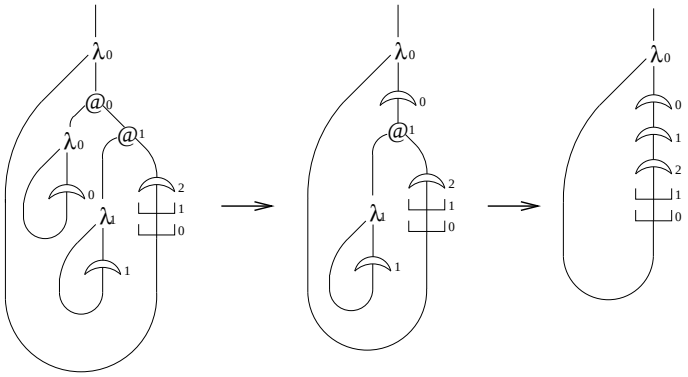


Fig. 9.2. The reduction of ($\underline{2}$ I)

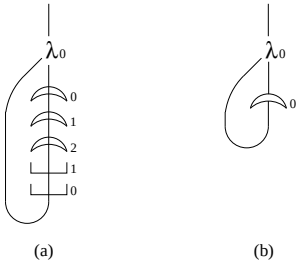


Fig. 9.3. Two representations of the identity

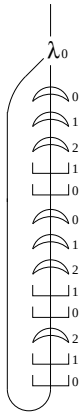


Fig. 9.4. Another representation of the identity in sharing graphs

reduction of the λ-term $\lambda n.(n\,\underline{2}\,I)$ is *exponential* in n (note that innermost reduction is linear!) and this exponential behavior is merely due to a problem of accumulation of "redundant" control operators (see the benchmarks at the end of this chapter).

A similar problem is caused by fans and garbage. Consider for instance the reduction of the term $I' = \lambda x.(\lambda z.\lambda w.w\ x\ x)$, represented in Figure 9.5 (note again, that $I' \to I$).

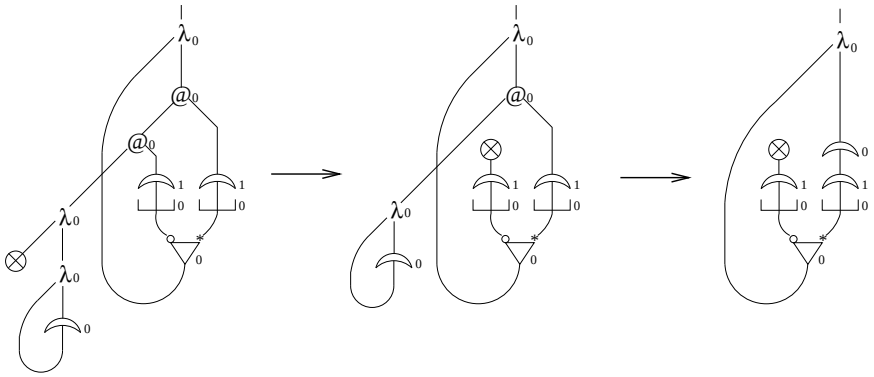


Fig. 9.5. The reduction of $\lambda x.(\lambda z.\lambda w.w\ x\ x)$

Even if we remove all redundant brackets and croissants, we still remain with the configuration in Figure 9.6, that contains a redundant garbage-fan pair.

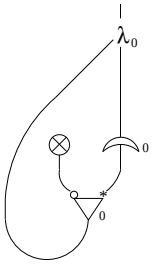


Fig. 9.6. Yet another graph for the identity

Again, these spurious nodes can easily accumulate, giving rise to an exponential explosion of the reduction time. For instance, in Figure 9.7 we give some benchmarks in BOHM (our prototyping implementation of Sharing Graphs) relative to the λ-term $\mathtt{f} = \lambda\mathfrak{n}.(\mathfrak{n}\ \underline{2}\ \mathrm{I}')$, where $\mathrm{I}' = \lambda\mathtt{x}.(\lambda\mathtt{z}.\lambda\mathtt{w}.\mathtt{w}\ \mathtt{x}\ \mathtt{x})$. Note the exponential explosion of the reduction time when the garbage collector is "off".

The first and obvious consequence of this fact is that garbage collection is *essential* in Lamping's algorithm, not only from the point of view of memory occupation, but also (and especially) for *efficiency* reasons. In particular, garbage collection should be performed *as soon as possible*. The problem is that the obvious garbage collection rules (that is, the rules collecting nodes by interaction of a garbage node with another form *at its principal port*) are not enough to solve the previous case. We shall discuss more about garbage collection later in this chapter and in the final Chapter.

9.2 Eliminating redundant information

We are thus interested in eliminating as much redundant information as possible.

Let us consider for instance the configuration in Figure 9.8.

We immediately realize two important properties of such a configuration:

(i) The pair of control operators can be always propagated together through other operators in the graph (that is, they can be regarded as a unique compound operator).

(ii) The total effect of this "compound" operator on the nodes they act upon is null (in the sense that they do not modify their level).

Input	G.C. on	G.C. off
(f one)	0.00 s. 30 interactions 2 fan int. 9 family red.	0.00 s. 31 interactions 5 fan int. 9 family red.
(f three)	0.00 s. 60 interactions 22 fan int. 15 family red.	0.00 s. 71 interactions 37 fan int. 15 family red.
(f five)	0.00 s. 98 interactions 42 fan int. 21 family red.	0.00 s. 155 interactions 105 fan int. 21 family red.
(f seven)	0.00 s. 136 interactions 62 fan int. 27 family red.	0.00 s. 383 interact. 317 fan int. 27 family red.
(f nine)	0.00 s. 174 interactions 82 fan int. 33 family red.	0.02 s. 1187 interactions 1105 fan int. 33 family red.
(f eleven)	0.00 s. 226 interactions 102 fan int. 39 family red.	0.08 s. 4309 interactions 4197 fan int. 39 family red.
(f thirteen)	0.00 s. 271 interactions 122 fan int. 45 family red.	0.18 s. 16640 interactions 16505 fan int. 45 family red.
(f fifteen)	0.00 s. 315 interactions 142 fan int. 51 family red.	0.88 s. 65834 interactions 65677 fan int 51 family red.

Fig. 9.7. Garbage collection: benchmarks for $f = \lambda n.(n \underline{2} I')$

Fig. 9.8. Croissant-bracket compound configuration

Recall now that the only problem of the optimal reduction technique is to guarantee the correct matching between fan-in and fan-out. Since two fans match if and only if they meet at the same level, and the "compound" configuration above has no total effect on levels, the natural idea would be to simply remove them from the graph, according to the rewriting rule in Figure 9.9.

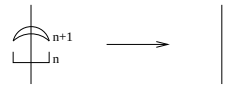


Fig. 9.9. Simplification of a croissant-bracket compound configuration

Unfortunately, the rule in Figure 9.9 is not correct in general. The reason is very simple: we forgot to take into account the possible annihilation of the innermost operator, as described by the critical pair in Figure 9.10.

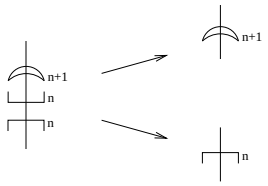


Fig. 9.10. Critical pair

Obviously, only the first reduction is correct, while the second one could easily lead to wrong and irreducible configurations.

Let us show that this can actually be a problem in practice. Consider the term

$$\lambda x.I(I(x\ P))\ I$$

where I is the identity and P is an arbitrary term. In Figure 9.11(a) we see the initial translation of this term.

By applying the rule in Figure 9.9 to this graph (see the dotted region), we obtain the graph in Figure 9.11(b). As we shall see, this simplification is harmless, here. Let us now proceed in the reduction. After firing three β-redexes we get the graph in Figure 9.12(a).

Now, the bracket traverses the λ (b) and the lower croissant (c). Finally, by firing the last β-redex we get our critical pair. In this case,

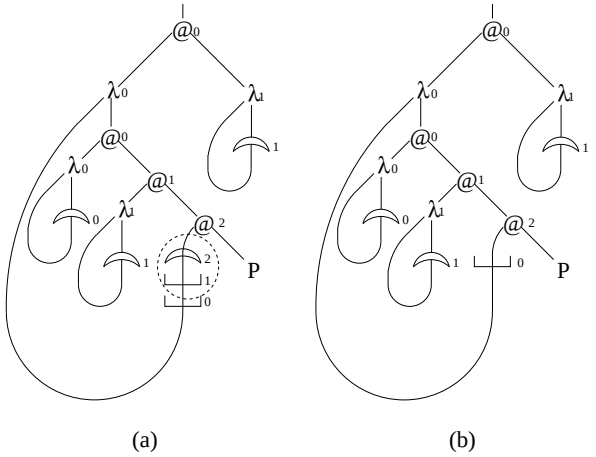


Fig. 9.11. Reduction of $\lambda x.I(I(x\,P))\,I\ldots$

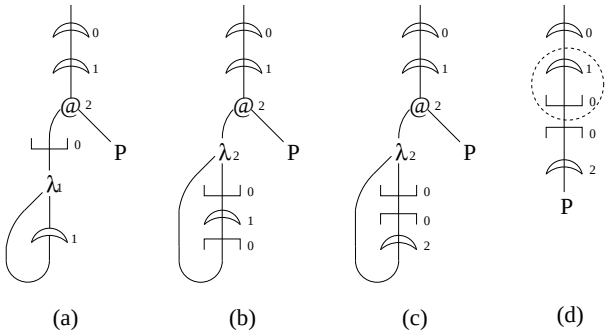


Fig. 9.12. ...ending with a critical pair

applying the rule of Figure 9.9 in the dotted region, we would get a deadlock between a bracket and a croissant at the same level.

Remark 9.2.1 A different way to understand the very problematic nature of the rule in Figure 9.9 is by noticing that its natural counterpart in dynamic algebra (namely, the rule $t!(d) = 1$) is *inconsistent*.
Indeed, suppose that $t!(d) = 1$. Since $t^* t = 1$, we have $!(d) = t^*$, and

$$\begin{aligned} !(t) &= t^* t!(t) \\ &= !(d)t!(t) \\ &= t!(!(d))!(t) \end{aligned}$$

$$\begin{aligned} &= t!(t^*)!(t) \\ &= t \end{aligned}$$

Then, let us take $u = !(r^*)t!(!(s))$. We have that $u = t!(!(r^*s)) = 0$. Moreover:

$$\begin{aligned} u &= !(r^*)!(t)!(!(s)) \\ &= !(r^*)(!(!(t))!(!(s)) \\ &= !(r^*)!(!(ts)) \\ &= !(!(ts))!(r^*) \end{aligned}$$

which implies

$$1 = !(!(ts))^*!(!(ts))!(r^*)!(r) = 0$$

A very similar problem is posed by fan and garbage nodes. In particular, we would be tempted to add the rewriting rule in Figure 9.13. One of the two branches of the fan-in has become garbage, and only the other one looks relevant.

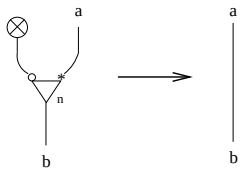


Fig. 9.13. Simplification of a fan-garbage compound configuration

However, in this case, the very "dangerous" nature of this rule should be clear. Again, the problem is that the fan-in could match with some other fan-out in the graph, for instance as in the example sketched in Figure 9.14.

In terms of paths, applying the above mentioned rule to the fan-garbage at the top in Figure 9.14 would cause the loss of the needed information about the correct "continuation" of the term at the matching fan at the bottom.

In the rewriting system, this corresponds to the critical pair in Figure 9.15.

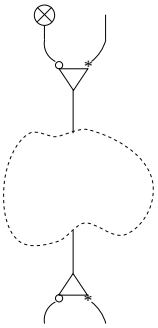


Fig. 9.14. Matching fans involving a fan-garbage compound pair

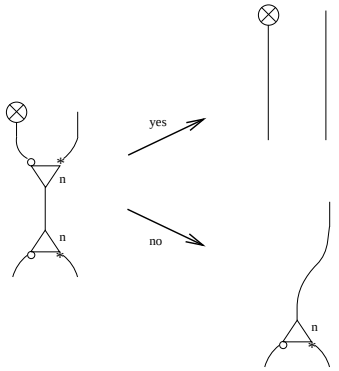


Fig. 9.15. Critical pair

9.3 Safe rules

In this section we shall give a theoretical characterization of the cases in which the simplification rules previously depicted can be safely applied. In the next section we will eventually see how to find a weaker notion of safeness applicable in practice.

We say that two nodes n_1 and n_2 in a sharing graph G *match* along a (consistent) path ϕ, if there exists a graph G' such that $G \twoheadrightarrow G'$ and the residual ϕ' of ϕ in G' is an annihilation configuration for (the residuals of) n_1 and n_2.

Definition 9.3.1 (safe) A sharing operator n (fan, croissant or bracket) in a sharing graph G is *safe* if, considering any consistent path of G, it can only match with itself.

Proposition 9.3.2 *The rules in Figure 9.16 are correct provided the lower operator is safe. Moreover, given the configurations in the rhs of rules 1,2,3,6, when the lower operator is safe, the upper one is safe too.*

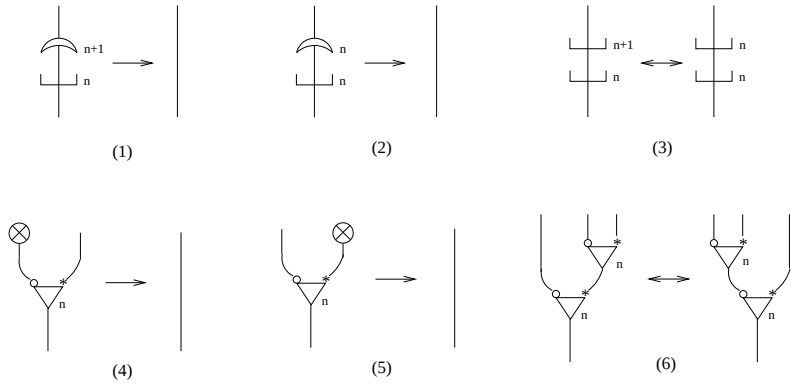


Fig. 9.16. Safe rules

Proof As we already noticed, the two operators in the right hand side of these rules can be always "propagated together"; more precisely, no other operator oriented in the same direction can ever appear inside the path connecting them. Let us note now that the only way to annihilate the upper operator is by previously annihilating the lower one (just check the levels). Since the lower operator is safe, it can only match with itself. As a consequence, also the upper operator can only match with itself; thus, it is safe too. Vice-versa, if the lower operator is annihilated, it can be eventually followed by an annihilation of the upper one. Finally, the configurations in the rhs and lhs of each rule change the levels of the operators they are propagated through in an identical way. So they cannot alter the matching of other operators in the graph, and hence neither the structure nor the consistency of paths. □

The interesting problem, that will be addressed to in the next section, is to find a simple and efficiently computable *sufficient condition* for safeness of control operator.

Remark 9.3.3 There is no natural way of orienting rules 3 and 6, above. This suggests that the configuration in the rhs of these rules should be more correctly reduced, respectively, to a unique new bracket

with "weight" 2 (and index n), and a unique new fan with three auxiliary doors (and index n). Actually, *all* control operators can be uniformly (and profitably) described as a unique "polymorphic" node (say multiplexer, see Chapter 7).

Remark 9.3.4 The reader with a categorical background should have no problem in recognizing in rules 1-3 above the three equations for the comonad "!". Indeed, we already established a relation between croissants and brackets and the two natural transformations $\epsilon_A : !(A) \to A$ and $\delta_A : !(A) \to !!(A)$ of the comonad "!". In fact, the equations characterizing the comonad are precisely the following:

$$\begin{aligned} !(\epsilon_A) \circ \delta_{!(A)} &= \mathrm{id}_{!(A)} \\ \epsilon_{!(A)} \circ \delta_{!(A)} &= \mathrm{id}_{!(A)} \\ !(\delta_{!(A)}) \circ \delta_{!(A)} &= \delta_{!!(A)} \circ \delta_{!(A)} \end{aligned}$$

Similarly, equations 3-6 describe the comonad structure of each exponential type $!(A)$. The interesting fact is that this analogy with category theory was actually the starting point for the study and the development of safe operators and their rewriting rules [Asp95].

9.4 Safe Operators

We shall now address the problem of recognizing safe operators. In particular, we shall give an easy mechanism for marking nodes with a suitable *safeness tag* (a boolean indicating if it is safe or unsafe). The tag is just a sufficient condition for safeness, i.e., a node with a tag safe is eventually safe, but a safe node could be tagged unsafe (see the example at the end of this section). The problem of giving a precise, operational characterization of safe nodes is still open. In spite of this fact, our tagging mechanism looks as a quite good approximation of safeness, and works pretty well on practical examples.

The algorithm for tagging nodes is the following:

(i) All sharing operators are initially tagged safe.
(ii) Both residuals of an operator interacting with a λ node are tagged unsafe.
(iii) All other interaction rules preserve the tag of the (ancestors of) the interacting operators.
(iv) Given the configurations in the lhs's of the rules in Figure 9.16, if the lower operator is tagged safe, the upper operator can be

tagged safe as well (it is interesting to observe that such configurations can be dynamically created along the reduction).

Definition 9.4.1 (t-safe) An operator is *t-safe* if it is tagged safe.

The rest of this section is devote to the proof of the following theorem.

Theorem 9.4.2 *If a sharing operator is t-safe, then it is safe.*

The proof is by induction on the length of the reduction.

The fact that all operators in the initial graph are safe is an immediate consequence of Corollary 6.6.5. Moreover, let us assume that at a given stage of the computation any t-safe operator is safe. Item 4 of the above tagging algorithm is an obvious consequence of Proposition 9.3.2. Hence, we left prove that the tagging algorithm is sound with respect to the application of any graph rewriting rule.

First of all, we see that no rule can change the safeness of an operator not involved in the rule, since consistent paths are preserved along the reduction. Then, let us start with the situations in which the proof is trivial.

When the interaction rule involves a λ-node, we have nothing to prove. At the same time, let us note that by our tagging algorithm, each t-safe operator is eventually "in" (e.g., a fan-in). In particular, since a sharing operator interacting with an application is "out" (e.g., a fan-out), it is eventually t-unsafe. Thus, also in the case of a rule involving an @-node we have nothing to prove.

When a safe operator interact with a croissant or a bracket, it obviously remains safe, by the uniqueness of the residual.

The only non trivial case is thus the case of a safe operator interacting with a fan. That is, in order to complete the proof we must show that:

Proposition 9.4.3 *Given the reduction in Figure 9.17, if the sharing operator* f *is t-safe before the reduction (and thus safe, by induction hypothesis), then both residuals of* f *are safe after the reduction.*

Since before the reduction in Figure 9.17 the node f was safe, safeness may be lost only in the case that, carrying on reducing, the residuals of f will mutually annihilate. Namely, the two residuals of f are unsafe only if there is a consistent path connecting them that contracts to an annihilation redex. In order to prove correctness of tagging, we have thus to prove that situations as the one depicted in Figure 9.18 cannot arise.

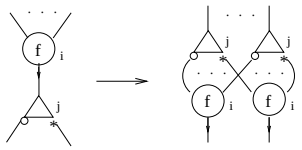


Fig. 9.17. Interaction between a fan and a safe node

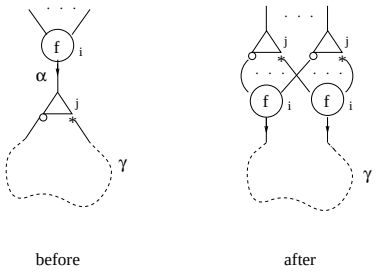


Fig. 9.18. The (inconsistent) configuration that could cause loss of safeness.

First of all, let us note that we can restrict to the case of a node that has never been t-unsafe. The idea is that any t-unsafe node has to touch some t-safe operator in order to reset its unsafe tag. Hence, we expect that either a node has always been t-safe (i.e., without any t-unsafe ancestor), or its principal port points (or better, has pointed) to some t-safe operator.

Let us say that a path $f_0 \dots f_k$ is a compound t-safe chain when: (*i*) any node in the path is t-safe; (*ii*) for $i = 0, \dots, k-1$, the principal port of f_i is connected to an auxiliary port of f_{i+i}; (*iii*) f_k is the only node in the chain that has always been t-safe. We see that the notion of compound chain of t-safe nodes is a sort of particular case of the lhs's of the rules in Figure 9.16: it is a sequence of them in which the front node has always been t-safe.

From the reduction point of view, we already remarked that such configurations can be seen as compound nodes. In fact, any propagation involving the first operator f_k in the chain $f_0 \dots f_k$ can be followed by the corresponding propagations of all the operators in the rest of the chain. This also explains why we could not be able to construct a t-safe chain moving from any t-safe operator f_i: the initial part $f_{i+1} \dots f_k$ of the chain might have interacted with some node, while the tail $f_0 \dots f f_i$ of the chain is still waiting to do so. Nevertheless, let us note that, when

the interaction was with a λ, all the chain is going to become unsafe, otherwise, when the interaction was with another control node, there is a reduction sequence involving control operators only that will recreate (a residual of) the original t-safe chain. (We invite the reader to formally prove the previous claim doing the next exercise.)

Exercise 9.4.4 Prove that, for any t-safe node in a sharing graph G, one of the following two cases holds:

(i) There is a reduction $G \twoheadrightarrow G'$ involving control nodes only such that any residual of f is contained in a chain $f_0 \ldots f_k$ of t-safe nodes in which f_k has always been t-safe.

(ii) There is a reduction $G \twoheadrightarrow G'$ involving control nodes only such that any residual of f is t-unsafe.

As a matter of fact, in the rest of the proof of Proposition 9.4.3, we shall this a property to restrict our attention to the case of a node that has always been t-safe.

So, let's go back to the configuration of Figure 9.18. Let α' and γ' be respectively the ancestors of α and γ, in the initial graph. Since f is t-safe, it did not traverse neither λ nor application nodes in α'. So, all *unmatched* λ and @ nodes must be oriented with their principal port towards the fan and α' must have the structure of Figure 9.19.

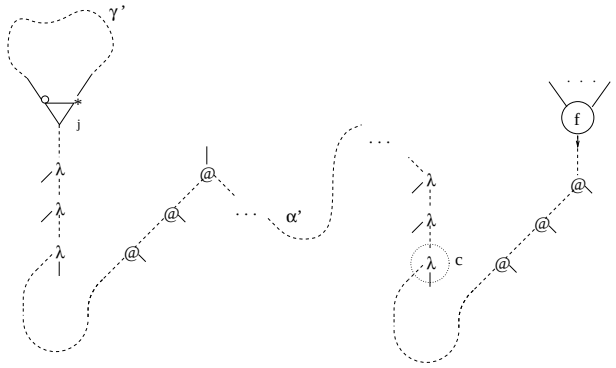


Fig. 9.19. Orientation of *unmatched* λ and @ nodes in α'.

In particular, for obvious polarity reasons, α' contains at least an unmatched λ-node. In the following, we shall consider the one closest to f, labeled with c in the previous picture.

Now, let us assume that that the residuals of f match along γ. Then f

matches with itself along $\alpha'\gamma'(\alpha')^r$. But, by the results on virtual interactions (section 6.6), we know that an operator in the initial graph can only match with itself along a path composed by a calling path, followed by an @-cycle and a return path (equal to the reverted calling path). Therefore, we have done if we prove that a path with this structure is inconsistent with the structure of $\alpha'\gamma'(\alpha')^r$.

The idea is simple: we have to prove that the λ marked with c must eventually belong to some calling path. Indeed, since c is unmatched in α' it must eventually be matched by an application in γ'. So, the fan belong to the calling path, and since each calling path must be eventually associated to an *equal* return path, we cannot return to the fan from a different auxiliary ports.

Proposition 9.4.5 *Let φ be an @-cycle, and ϕ be an initial segment of φ terminating at the bound port of a λ-node l which is unmatched in ϕ (we consider ϕ as a directed path). If it exists an initial context for ϕ that does not contain any # symbol, then either l is the initial node of a calling path or l was previously traversed by ϕ passing from its principal to its body port (the exiting port is forcedly the body port, due to the assumptions on the context).*

Proof By induction on the definition of @-cycle. If the @-cycle is elementary, the statement is obvious (due to the assumptions on the context, the only way we have to reach a variable is by traversing its binder). Let us come to the inductive case. Suppose that we reach a free variable inside the argument N of the application connected to some λ-node l. If l is unmatched, we have done, since we must have here a calling path by definition of @-cycle. Otherwise, l is matched, so we jump over the calling path, and enter another @-cycle. We can now apply the induction hypothesis for this @-cycle (indeed no # symbol can have been added to the context, so far). So there exists a node l' with the desired properties w.r.t. the internal @-cycle. There are two possibilities: either l' is unmatched in ϕ (and then we have done) or it is matched. In the latter case, it is eventually matched inside the argument of the external application. This means that we eventually complete the internal cycle, come back inside N, exit from the argument of the matching application and start traveling down again. If we reach another free variable of N, we repeat the reasoning (again, no # symbol has been added). Otherwise, we reach a variable bound inside N, but then we forcedly traversed

its binder $\mathfrak{l}$ from the principal to the body port, since this is the only way the path could enter inside the body of $\mathfrak{l}$. □

Corollary 9.4.6 *Let φ be an @-cycle, and ϕ be an initial segment of φ terminating at the bound port of a λ-node $\mathfrak{l}$ which is unmatched in ϕ. If ϕ does not traverse unmatched λ-nodes then $\mathfrak{l}$ is the initial node of a calling path.*

Proof If ϕ does not traverse unmatched λ-nodes we do not need any # symbol in the initial context. Then apply the previous proposition. □

For an alternative and more formal proof of the correctness of our tagging technique (using proper paths a la Lamping), the reader may consult [AC96].

9.5 Examples and Benchmarks

The aim of this section is to show the practical relevance of the safe rules by comparing, on a few simple examples the relative performances of the rewriting system with and without them. As you will see, the introduction of the safe rules is actually something more than a mere "optimization": in some cases it makes the reduction time linear instead of exponential. With these additional rules (and only with them), Lamping's technique becomes really competitive with respect to more traditional implementation techniques.

The benchmarks refers to the Bologna Optimal Higher-order Machine (BOHM), our prototype implementation of (a variant of) Lamping's graph reduction technique. Although BOHM's source language constitutes the full core of a typical dynamically-typed lazy functional language, including a few data types, conditional expressions, recursive values and lazy pattern matching, we will mainly present examples from pure λ-calculus, here, since these terms are particularly well suited to investigate the problem of accumulation of control operators. Moreover, in order to give a gist of optimality and its actual power, we shall also compare BOHM's performance with two fully developed and largely diffused implementations: Caml Light 0.5 and Yale Haskell Y2.3b-v2 (see Chapter 12 for other "real word" benchmarks).

Our first example is a "primitive recursive" version of the factorial function (on Church integers).

```
def Succ = \n.\x.\y.(x (n x y));;
```

```
def Mult = \n.\m.\x.(n (m x));;
def Pair = \x.\y.\z.(z x y);;
def Fst = \x.\y.x;;
def Snd = \x.\y.y;;
def Nextfact = \p.let n1 = (p Fst) in
                  let n2 = (Succ (p Snd)) in
                      (Pair (Mult n1 n2) n2);;
def Fact = \n.(n Nextfact (Pair one zero) Fst);;
```

Since the evaluation in BOHM stops at weak head normal forms, we have to supply enough arguments (identities, for instance) to get an interesting computation. The result of the test is in Figure 9.20. The first column of the table is the input term. The two following columns respectively refer to the optimal implementation without the safe rules (BOHM 0.0), and with them (BOHM 1.1). The last columns refer to Caml Light and Haskell, respectively. Each entry in the table contains the user time required for reducing the input by each system (on a Sun Sparcstation 5). In the case of optimal systems, we also give the number of fan interactions (a measure of the complexity of the abstract algorithm), as well as the total number of interactions (i.e., of elementary operations) executed during the computation, and the length of the family reduction. The latter datum is the number of redex-families (i.e., *optimally shared* β-reductions) required for normalizing the term. This is the same as the total number of interactions between application and lambda nodes in the graph. These data are slightly different in the two versions of BOHM, due to a different management of globally defined terms (see Chapter 12).

The improvement provided by the safe rules is amazing. The old version explodes for input seven, Caml Light for input ten, while with the new version we compute the factorial of twenty in 0.33 seconds.

Unfortunately, things are not always that good for BOHM. Our next example is a computation of the Fibonacci sequence, defined as follows:

```
def Add = \m.\n.\x.\y.(m x (n x y));;
def Nextfibo = \p.let n1 = (p Fst) in
                  let n2 = (p Snd) in
                  (Pair (Add n1 n2) n1);;
def Fibo = \n.(n Nextfibo (Pair zero one) Fst);;
```

The result of the test is in Figure 9.21. Again, there is a substantial

Input	BOHM 0.0	BOHM 1.1	Caml Light	Haskell
(fact one I I)	0.00 s. 561 interactions 71 fan int. 20 family red.	0.00 s. 133 interactions 16 fan int. 30 family red.	0.00 s.	0.00 s.
(fact three I I)	0.04 s. 3375 interactions 251 fan int. 45 family red.	0.00 s. 386 interactions 154 fan int. 55 family red.	0.00 s.	0.01 s.
(fact five I I)	0.23 s. 17268 interactions 412 fan int. 74 family red.	0.00 s. 922 interactions 341 fan int. 84 family red.	0.02 s.	0.05 s.
(fact seven I I)	explodes	0.02 s. 1952 interactions 608 fan int. 117 family red.	0.17 s.	0.84 s.
(fact nine I I)		0.03 s. 3820 interactions 979 fan int. 154 family red.	10.60 s.	explodes
(fact ten I I)		0.05 s. 5202 interactions 1211 fan int. 174 family red.	explodes	
(fact twenty I I)		0.33 s. 53605 interactions 5983 fan int. 435 family red.		

Fig. 9.20. Factorial

improvement with respect to the old version, but in this case Caml Light is still more efficient.

Note that the total number of family reductions grows quadratically in the case of the factorial function while it grows as Fibonacci for the Fibonacci function. In the first case, the optimal compiler is able to profit of the big amount of sharing, obtaining very good performance on big input values, while in the second case the computation is already intrinsically exponential, and Caml Light takes advantage of its simpler abstract reduction model, avoiding book-keeping operations that in

Input	BOHM 0.0	BOHM 1.1	Caml Light	Haskell
(fibo one I I)	0.01 s. 513 interactions 72 fan int. 20 family red.	0.00 s. 121 interactions 14 fan int. 28 family red.	0.00 s.	0.00 s.
(fibo four I I)	0.06 s. 3152 interactions 244 fan int. 55 family red.	0.00 s. 462 interactions 174 fan int. 63 family red.	0.00 s.	0.01 s.
(fibo seven I I)	0.19 s. 12749 interactions 513an int. 98 family red.	0.00s. 1260 interactions 440 fan int. 106 family red.	0.01 s.	0.01 s.
(fibo ten I I)	0.74 s. 52654 interactions 1260 fan int. 173 family red.	0.03 s. 3898 interactions 1178 fan int. 181 family red.	0.02 s.	0.04 s.
(fibo thirteen I I)	3.08 s. 230548 interactions 4023 fan int. 390 family red.	0.12 s. 14357 interactions 3916 fan int. 398 family red.	0.04 s.	0.15 s.
(fibo sixteen I I)	explodes	0.38 s. 57844 interactions 15126 fan int. 1185 family red.	0.12 s.	0.85 s.
(fibo nineteen I I)		1.57 s. 241291 interactions 62224 fan int. 4412 family red.	0.44 s.	3.69 s.

Fig. 9.21. Fibonacci

this case are useless. Note in particular that the execution time in the Caml Light system is *linear* in the number of family reductions (i.e., optimal in the strongest sense) in the case of Fibonacci, so this term is surely *one of the best examples we could choose for Caml Light.*

Having considered an example particularly favorable to Caml Light and other standard implementation models, let us now see a couple of examples particularly suited to Lamping's technique.

Let us first consider the term $\mathfrak{g} = \lambda\mathfrak{n}.(\mathfrak{n}\,\underline{2}\,\mathrm{I}\,\mathrm{I})$, where $\underline{2}$ is the Church numeral representing the number 2, and I is the identity. In this case (see

Figure 9.22) the (new version of the) optimal interpreter works linearly w.r.t. its input, while Caml Light and Haskell are exponential (just like the old implementation).

Input	BOHM 0.0	BOHM 1.1	Caml Light	Haskell
(g one)	0.00 s. 124 interactions 23 fan int. 7 family red.	0.00 s. 30 interactions 4 fan int. 7 family red.	0.00 s.	0.00 s.
(g four)	0.00 s. 608 interactions 62 fan int. 16 family red.	0.00 s. 79 interactions 34 fan int. 16 family red.	0.00 s.	0.01 s.
(g seven)	0.04 s. 3052 interactions 101 fan int. 25 family red.	0.00 s. 136 interactions 64 fan int. 25 family red.	0.00 s.	0.09 s.
(g ten)	0.43 21176 interactions 140 fan int. 34 family red.	0.00 s. 193 interactions 94 fan int. 34 family red.	0.03 s.	0.70 s.
(g thirteen)	explodes	0.00 s. 280 interactions 126 fan int. 49 family red.	0.20 s.	6.12 s.
(g sixteen)		0.00 s. 346 interactions 156 fan int. 58 family red.	1.60 s.	explodes
(g nineteen)		0.00 s. 412 interactions 186 fan int. 67 family red.	12.65 s.	

Fig. 9.22. $g = \lambda\, n.(n\ \underline{2} I\ I)$

Things are even more impressive if we consider the λ-term $f = \lambda n.(n\ \underline{2}\ \underline{2}\ II)$. Notice that, in this case, the number of fan interactions is exponential in the number of family reductions: this is therefore a case in which the abstract algorithm is exponential in Lévy's complexity measure.

Input	BOHM 0.0	BOHM 1.1	Caml Light	Haskell
(f one)	0.00 s. 282 interactions 42 fan int. 12 family red.	0.00 s. 50 interactions 22 fan int. 12 family red.	0.00 s.	0.00 s.
(f two)	0.00 s. 765 interactions 67 fan int. 17 family red.	0.00 s. 94 interactions 44 fan int. 17 family red.	0.00 s.	0.02 s.
(f three)	0.04 s. 7001 interactions 100 fan int. 22 family red.	0.00 s. 170 interactions 74 fan int. 22 family red.	0.00 s.	0.18 s.
(f four)	explodes	0.00 s. 346 interactions 120 fan int. 27 family red.	1.02 s.	53.01 s.
(f five)	explodes	0.00 s. 866 interactions 198 fan int. 32 family red.	explodes	explodes
(f six)		0.00 s. 2650 interactions 340 fan int. 37 family red.		
(f ten)		3.58 s. 531706 interactions 4236 fan int. 57 family red.		

Fig. 9.23. $f = \lambda\ n.(n\ \underline{22}I\ I)$

10
Complexity

The first results about the inherent complexity of optimal reduction of λ-expressions are very recent, and this topic is still full of many interesting open problems. We currently know that the cost of parallel β-reduction is not bounded by any Kalmar-elementary recursive function [AM97]. This means that the parallel β-step is not even remotely a unit-cost operation: the time complexity of implementing a sequence of n parallel β-steps is not bounded as $O(2^n)$, $O(2^{2^n})$, $O(2^{2^{2^n}})$, or in general, $O(\mathbf{K}_\ell(n))$ where $\mathbf{K}_\ell(n)$ is a fixed stack of ℓ 2's with an n on top.

A key insight, essential to the establishment of this nonelementary lower bound, is that any simply-typed λ-term can be reduced to normal form in a number of parallel β-steps that is only linear in the length of the explicitly-typed term. The result then follows from Statman's theorem that deciding equivalence of typed λ-terms is not elementary recursive. This theorem gives a lower bound on the work that must be done by *any* technology that implements Lévy's notion of optimal reduction. So, we cannot expect a miracle from Lamping's algorithm. Yet this sharing technology remains a leading candidate for correct evaluation without duplication of work. In this case, the close complexity analysis of sharing graph implementation just improve our appreciation of its very parsimonious way of handling sharing, even if parallel β-steps are *necessarily* resource-intensive.

10.1 The simply typed case

A key insight, essential to the establishment of our nonelementary lower bound, is that any simply-typed λ-term can be reduced to normal form in a number of parallel β-steps that is only linear in the length of the explicitly-typed term. The proof of this claim depends on the judicious

use of η-*expansion* to control the number of parallel β-steps. Not only does η-expansion act as an *accounting mechanism* that allows us to see order in the graph reduction, it moreover serves as a sort of *optimizer* that exchanges the work of parallel β-reduction for the work of *sharing*.

Remarkably, we shall prove the previous result, that does not concern *any* specific implementation, by making a very simple use of Lamping's technology. In particular, since we already know that this graph reduction method is algorithmically correct, it lets us work out calculations that would be virtually impossible, and certainly inscrutable, in the labelled λ-calculus. In our opinion, this mere proof is already enough to offer a good justification and a solid theoretical status to sharing graphs.

10.1.1 Typing Lamping's rules

As we already observed several times, Lamping's graph rewriting rules can be naturally classified in two main groups:

(i) the rules involving application, abstraction and sharing nodes (fan), that are responsible for β-reduction and duplication (this is what we call the *abstract algorithm*);
(ii) some rules involving control nodes (square brackets and croissants), which are merely required for the correct application of the first set of rules.

More precisely, the first set of rules requires an "oracle" to discriminate the correct interaction rule between a pair of fan-nodes; the second set of rules can be seen as an effective implementation of this oracle. This distinction looks particularly appealing since *all* different translations proposed in the literature after Lamping [GAL92a, GAL92b, Asp94, Asp95] (see Chapter 8) differ from each other just in the way the oracle is implemented (in the sense that all of them perform *exactly* the same set of abstract reductions).

We are not concerned with the oracle, here, since it does not play any role in our proof (you may just suppose the existence of an oracle, and not even an *effective* one). Now, in the simply typed case, Lamping's abstract rules assume the shape in Figure 10.1.

The general idea is that each edge† of the graph is labelled with a suitable type (in the pictures, we shall usually use the notation β^α in-

† Instead of typing edges, we could equivalently type each port of a node (adding, moreover, a suitable polarity). Our approach has been essentially adopted for typographical reasons.

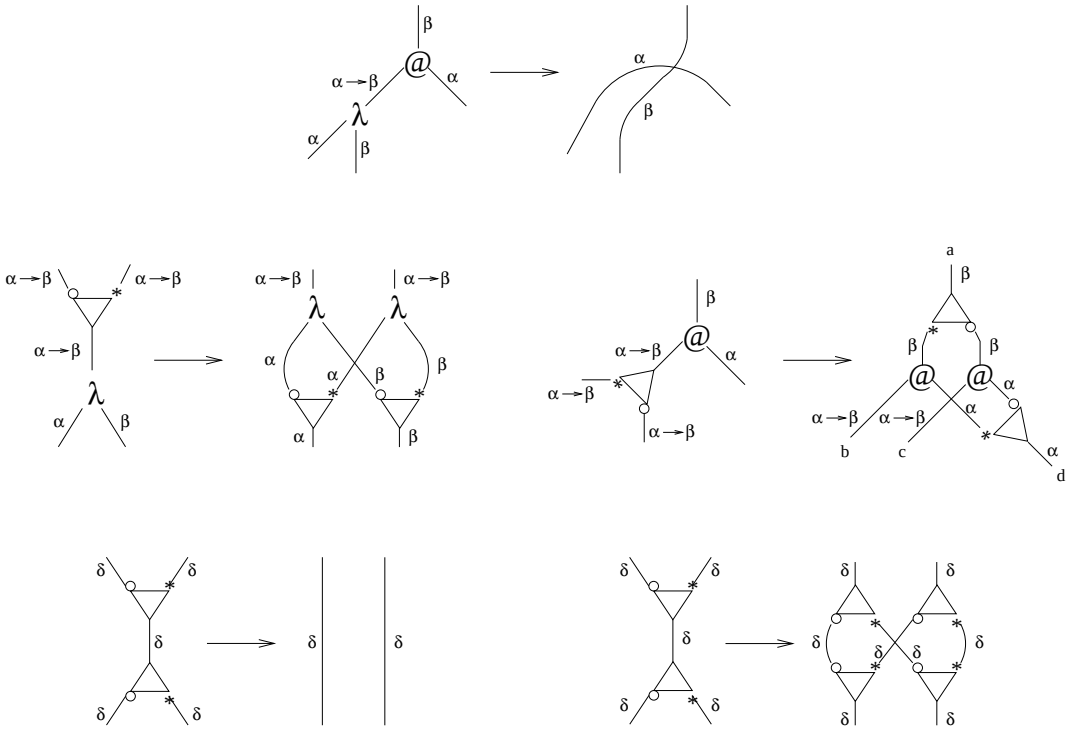


Fig. 10.1. Lamping's simply typed rules.

stead of $\alpha \to \beta$, since it is more compact). Let us remark that types do not play any role in the reduction of the term; they are just an obvious invariant of the computation. A graph is well typed if and only if for each node n in the graph, the types of the edges connected to n satisfy the constraints imposed in Figure 10.2.

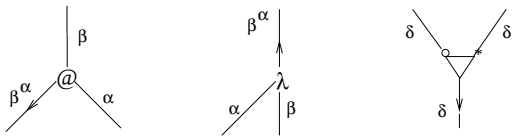


Fig. 10.2. Well typed sharing nodes.

Definition 10.1.1 (type of a node) The *type* of a node is the type of the edge connected to its principal port.

Now, we can already state the main, straightforward but crucial property of the simply typed case:

Proposition 10.1.2 *The type of fan-nodes may only decrease along the reduction.*

Proof Just look at the typed interaction rules of Figure 10.1. □

In other words, the flow of fan nodes is always from regions of higher type to regions of equal or lower type.

10.1.2 The η-expansion method

Given a simply typed λ-term E, we show how to construct a variant E' that is βη-equivalent to E, derived by introducing η-*expansions* of bound variables in E. The size of E', and the size of the initial sharing graph that represents E', are larger than E by only a small constant factor. Moreover, the number of parallel β-steps needed to normalize E' is linearly bounded by its size. As a consequence, we demonstrate that the normal form of any simply typed λ-term can be derived in a linear number of parallel β-steps. In order to make these calculations more precise, we need to define what we mean by the *size* of types, λ-terms, and graphs:

Definition 10.1.3 (size) We define the *size of a simple type* by structural induction:

$$\begin{aligned} \|o\| &= 1 \\ \|\alpha \to \beta\| &= 1 + \|\alpha\| + \|\beta\|. \end{aligned}$$

Similarly, we inductively define the *size of a simply-typed term* as:

$$\begin{aligned} |x| &= \|\sigma\| \qquad \text{if } x \text{ has type } \sigma \\ |\lambda x : \sigma . E| &= 1 + |E| \\ |(EF)| &= 1 + |E| + |F|. \end{aligned}$$

The number $|E|$ simply counts 1 for each λ and apply in E, plus $\|\sigma\|$ for each variable of type σ. Finally, we refer to the *size* $[\![G]\!]$ *of a sharing graph* G as the number of its nodes.

The only unusual feature of this definition is that the size of a variable is given by the size of its type. Had we instead used the more usual

definitions $|x| = 1$ and $|\lambda x : \sigma.E| = 1 + \|\sigma\| + |E|$, the size of terms would be polynomially smaller, but only by a quadratic factor.

The bound we prove on the number of parallel reductions depends essentially on controlling the duplication of λ and apply nodes by sharing nodes. When a sharing node has type $\alpha \to \beta$ and faces the principal port of either a λ-node or an apply node, duplication creates two sharing nodes, of types α and β respectively. If the value being shared by a node is the base type o, then that sharing node cannot interact with a λ or apply node, since the principal ports of those nodes cannot sit on wires that are at base type—they are functions.

As a consequence, each sharing node has a *capacity* for self-reproduction that is bounded by the size of the type of the value being shared. The idea of introducing η-*expansion* is to force a node sharing a value x of type σ to the base type o, by making that node duplicate components of the graph coding the η-expansion of x. This technique leads to an efficient *accounting mechanism* which bounds the duplication of λ and apply nodes in a graph reduction, and hence bounds the number of parallel β-steps. In addition, it serves as a lovely *optimization* method, where parallel β-reduction is simulated by the interaction between sharing nodes *only*.

Definition 10.1.4 (η-expansion) Let x be a variable of type σ. The η-*expansion* $\eta_\sigma(x)$ of x is the typed λ-term inductively defined on σ as follows:

$$\begin{aligned} \eta_o(x) &= x \\ \eta_{\alpha_1 \to \cdots \to \alpha_k \to o}(x) &= \lambda y_1 : \alpha_1. \ \ldots \ \lambda y_k : \alpha_k.x\,(\eta_{\alpha_1}(y_1)) \ldots\, (\eta_{\alpha_k}(y_k)) \end{aligned}$$

In the graph representation, each η-expanded variable is coded by a subgraph with two distinguished wires that we respectively call the *positive entry* and the *negative entry* of the variable (see Figure 10.3). When the variable is of base type o, this subgraph is just a wire.

The graph representing an η-expanded term has the following nice properties:

Lemma 10.1.5 *If G is the initial sharing graph representing $\eta_\sigma(x)$, then $[\![G]\!] \leq 2\|\sigma\|$.*

Proof By induction on σ. Notice that if $\sigma \equiv \alpha_1 \to \alpha_2 \to \cdots \alpha_k \to o$, then $[\![G]\!] = 2k + \sum_{1 \leq i \leq k} [\![G_{\alpha_i}]\!]$, where G_{α_i} represents $\eta_{\alpha_i}(y_i)$. □

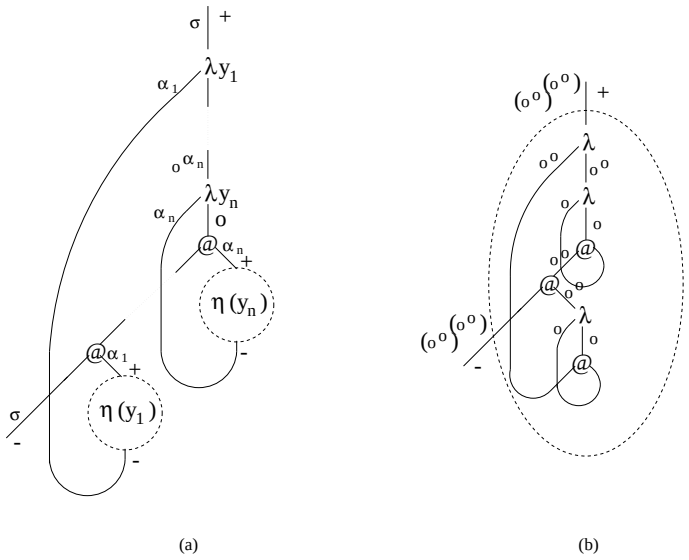


Fig. 10.3. (a) η-expansion; (b) $\eta_{(o\to o)\to o\to o}(x)$.

Lemma 10.1.6 *Let* x *be a variable of type* σ*, and let* G *be the graph representing* $\eta_\sigma(x)$*. Then in an optimal reduction, any sharing of* G *at the positive or negative entry results in a residual graph* $\Delta(x)$ *where all copies of the sharing node are at base type, and* $[\![\Delta(x)]\!] \leq 3\|\sigma\|$.†

Proof The proof is a simple induction on σ. If $\sigma = o$, the conclusion is immediate and trivial—no reductions are possible, because G is just a single wire.

Suppose $\sigma = \alpha_1 \to \ldots \to \alpha_n \to o$. Then the graph G coding $\eta_\sigma(x)$ has the structure depicted in Figure 10.4(a), with n λ-nodes and n apply nodes. If a sharing node is placed at the *positive* entry, that node will duplicate the n λ-nodes; copies of the sharing node will move to the auxiliary (non-interaction) port of the top apply node, which has base type o, and to the *negative* entries of the subgraphs G_i representing $\eta_{\alpha_i}(y_i)$. Dually, if a sharing node is placed at the *negative* entry, that node will duplicate the n apply nodes; copies of the sharing node will move to the auxiliary port of the top λ-node, which has base type o,

† We chose the notation $\Delta(x)$ to remind the reader that the graph is defined by propagating the sharing nodes (hence the Δ) to base type.

and to the *positive* entries of the subgraphs G_i representing $\eta_{\alpha_i}(y_i)$. The lemma follows by induction on the α_i. □

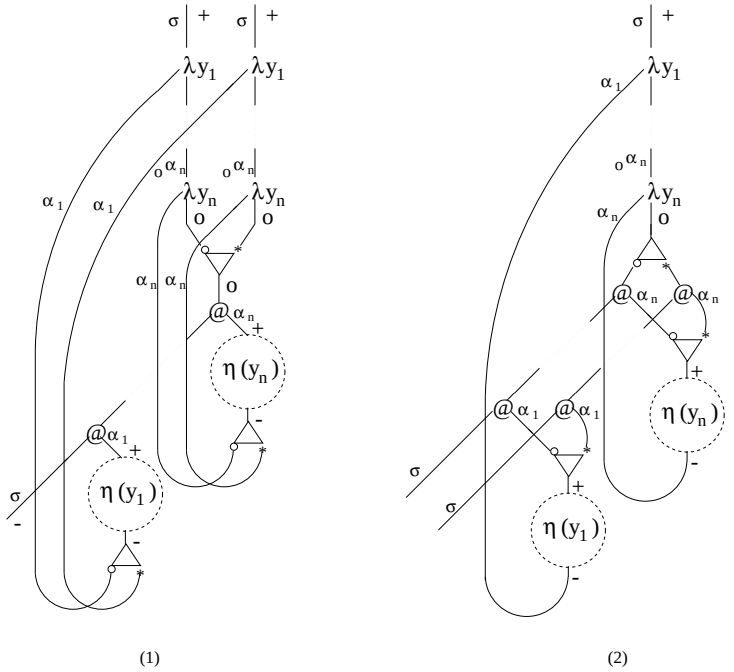


Fig. 10.4. Fan propagation inside $\eta(x)$.

Example 10.1.7 The graph in Figure 10.5 shows the duplication of $\Delta(\eta_{(o\to o)\to o\to o}(x))$ by a single sharing node.

The previous lemma may be easily generalized to an arbitrary tree network of sharing nodes, or equivalently, the t-fold *multiplexers* of Guerrini [Gue96]. In this case, the duplicated "skeleton" of the sharing graph representing $\eta_\sigma(x)$ is replicated for each instance of its use by the t leaves of the multiplexor.

Corollary 10.1.8 *Let $\Delta^t(x)$ be the residual graph that results from the sharing of the graph representing $\eta_\sigma(x)$ by a binary tree of t sharing nodes. Then $[\![\Delta^t(x)]\!] \leq (2+t)\|\sigma\|$.*

In the initial sharing graph coding a λ-term $\lambda x : \sigma.E$, multiple references to the bound variable are represented by exactly such a tree

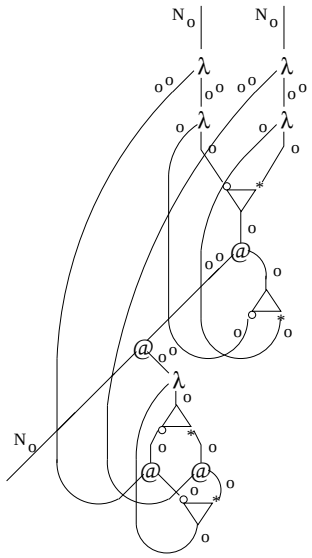


Fig. 10.5. Duplication of $\eta_{(o\to o)\to o\to o}(x)$.

network of sharing nodes. This tree is connected to the (auxiliary) parameter port of the λ-node that represents the binding. Because this parameter port is not a primary port, no interaction is possible between the sharing nodes and the λ-node; the sharing nodes are "stuck" until the λ-node is annihilated in a parallel β-reduction. We seek to control the possible node replication that could result from such a reduction, by forcing the λ-term $\lambda x : \sigma.E$ to be applied to $\eta_\sigma(x')$. We may then conclude that the duplication caused by a sharing node is proportional to a fixed function of the size of the initial term, and is *not* affected by the size of intermediate terms in the reduction sequence. In other words, the duplication is amenable to control via static analysis of the initial term. These intuitions are clarified in the following definition.

Definition 10.1.9 (optimal root) Let M be a simply-typed λ-term. The *optimal root* $\mathbf{or}(M)$ of M is derived by replacing every subterm of the form $\lambda x : \sigma.E$ with $\lambda x' : \sigma.(\lambda x : \sigma.E\ \eta_\sigma(x'))$, where $\sigma \neq o$ and x occurs more than once in E. We refer to the new β-redexes introduced by this transformation as *preliminary redexes*.

It should be clear that the transformation is applied at most once to any subterm $\lambda x : \sigma.E$. Since $\mathbf{or}(M)$ is obtained by M by means of η and

β-expansions, we also know that $M =_{\beta\eta} \mathbf{or}(M)$. The transformation can be understood as duplicating, once and for all, the "skeleton" of this term that describes all of its possible uses. The relevant information about these uses is provided unambiguously by the type σ.

Definition 10.1.10 We define $\Delta(M)$ to be the sharing graph obtained from $\mathbf{or}(M)$ by reducing all of the preliminary redexes, and propagating all sharing nodes to the base type.

We emphasize the following crucial property of $\Delta(M)$:

Lemma 10.1.11 *All sharing nodes in $\Delta(M)$ have atomic types.*

Proof After the β-reduction of all preliminary redexes, all sharing nodes are positioned to interact at the positive entry of the subgraphs representing η-expanded variables. Following these essentially structural reductions, the sharing nodes duplicate these subgraphs. Lemma 10.1.6 ensures that all such sharing nodes can propagate to base type. □

Definition 10.1.12 (trivial) A bound variable is *trivial* in a typed λ-term if it is at base type, or occurs at most once. A typed λ-term is *trivial* if all of its bound variables are trivial.

Lemma 10.1.13 *Let $F \equiv \lambda x : \sigma.E$ be a nontrivial term where E is trivial. Then $[\![\Delta(F)]\!] \leq 2|F|$.*

Proof Let G be the sharing graph coding $\mathbf{or}(F) \equiv \lambda x' : \sigma.F(\eta_\sigma(x'))$, and let G' be derived by graph reduction of the outermost β-redex, representing the term $\lambda x' : \sigma.E[\eta_\sigma(x')/x]$.
Assume x occurs t times in E, where G_E is the sharing graph representing E, so that the sharing of $\eta_\sigma(x')$ is represented in G' by a tree of sharing nodes with t leaves. Since $[\![G_E]\!]$ counts only the number of occurrences of λ and apply in E, it should be clear that $[\![G_E]\!] \leq |E| - t\|\sigma\|$.
To construct $\Delta(F)$ from G', we need only propagate the sharing nodes into the graph representation of $\eta_\sigma(x')$, generating $\Delta^{t-1}(x')$ with size at most $(1+t)\|\sigma\|$, by Corollary 10.1.8. Then the size of $\Delta(F)$ is

$$[\![\Delta(F)]\!] \leq 1 + (|E| - t\|\sigma\|) + (1+t)\|\sigma\| = 1 + |E| + \|\sigma\| = |F| + \|\sigma\|.$$

However, $\|\sigma\| \leq |F|$, since $\|\sigma\|$ just counts the contribution of one occurrence of x in F; we then conclude $[\![\Delta(F)]\!] \leq 2|F|$. □

Theorem 10.1.14 *Let* F *be a simply-typed* λ*-term. Then* $[\![\Delta(F)]\!] \leq 2|F|$.

Proof The proof is just a generalization of the previous lemma. We construct $\Delta(F)$ by insertion of the same preliminary redexes, reduction, and propagation of sharing nodes through the graphs representing $\eta_\alpha(x)$ over variables x of type α. Assume without loss of generality that the name of each bound variable is unique; then by Corollary 10.1.8,

$$\begin{aligned}
[\![\Delta(F)]\!] &\leq \left(|F| - \sum_{\substack{x\,:\,\sigma \\ x \text{ nontrivial}}} \mu(x)\|\sigma\|\right) + \left(\sum_{\substack{x\,:\,\sigma \\ x \text{ nontrivial}}} (1+\mu(x))\|\sigma|\right) \\
&\leq |F| + \left(\sum_{\substack{x\,:\,\sigma \\ x \text{ nontrivial}}} \|\sigma\|\right),
\end{aligned}$$

where $\mu(x)$ is the number of occurrences of x in F. When $\mu(x) \geq 2$, only $\mu(x) - 1$ sharing nodes are needed in the initial graph representation. Again, it is clear that $\sum_x \|\sigma\| \leq |F|$, so that $[\![\Delta(F)]\!] \leq 2|F|$. □

The reader may be bothered by this "linear" bound, which depends on the definition of the size function $|F|$, where the occurrence of an x of type σ contributes $\|\sigma\|$ to the sum. Suppose we had instead chosen the definition of size as

$$\begin{aligned}
\Lambda(x) &= 1 \\
\Lambda(\lambda x : \sigma.E) &= 1 + \|\sigma\| + \Lambda(E) \\
\Lambda(EF) &= 1 + \Lambda(E) + \Lambda(F),
\end{aligned}$$

so that $\Lambda(E)$ is the length of the explicitly-typed term. Since it is not hard to show that $|M| \leq \Lambda(M)^2$, we would derive instead $[\![\Delta(F)]\!] \leq 2\Lambda(F)^2$ as the statement of the previous theorem. The significance of either inequality is that $[\![\Delta(F)]\!]$ *is only polynomial in* $|F|$, which is a good enough bound to derive our more important results.

As an obvious consequence of Lemma 10.1.11, we have the following important observation.

Theorem 10.1.15 *The total number of* β*-reductions (and thus of families) in the graph normalization of* $\Delta(M)$ *cannot exceed its initial size.*

Proof Since all sharing nodes have atomic types in $\Delta(M)$, they cannot interact with abstractions or applications. As a consequence, no new application or λ-node can be created during the reduction. Since

β-reduction can only make the graph *smaller* via annihilation of complementary λ and apply nodes, the total number of β-reductions is bounded by the initial size of the graph $\Delta(M)$. □

If $\Delta(M)$ is considered as the representation of a logical proof via the Curry-Howard analogy, with the caveat that some sense is made of *fan-out*, it gives a very interesting "normal" form, where all logical rules are "below" the structural ones. This provides for some computational amusement: one can immediately and easily remove all the logical cuts. The rest of the computation is merely structural—the annihilation or duplication of sharing nodes.

A fundamental consequence of the bound on the number of redex families in $\mathbf{or}(M)$ is its *strong normalization*, as well as the strong normalization of M). This observation is a trivial conseqeunce of Theorem 10.1.15.

Theorem 10.1.16 *Let Σ be any finite (possibly parallel) reduction of a term M. Then any reduction Σ'* relative† *to Σ is terminating.*

Since all redexes created along *any* reduction of $\mathbf{or}(M)$ eventually belong to some of its families, any reduction strategy is terminating.

Theorem 10.1.17 *The simply typed λ-calculus is strongly normalizing.*

Even more, the bound on the reduction is, if one counts parallel β-reduction of families, merely linear in the length of the initial term.

10.1.3 Simulating generic elementary-time bounded computation

Now that we know that the normal form of a simply-typed λ-term can be computed in a linear number of parallel β-steps, the next goal is to construct a *generic reduction* from the largest time hierarchy we can manage via a *logspace reduction.*‡ For example, if we can simulate deterministic computations in DTIME$[2^n]$ (deterministic exponential time) in

† Σ' is relative to Σ if all redexes in Σ' are in the same family of some redex in Σ, see Chapter 5.

‡ This *reduction* is just a compiler that takes an arbitrary Turing Machine M running in some time bound $t(n)$ on an input x of size n, and produces a typed λ-term $e : \mathsf{Bool}$ such that e reduces to the term coding **true** iff M accepts x in $f(n)$ steps. The "logspace" means that the compiler has only $O(\log n)$ bits of internal memory to carry out the compilation, ensuring that the output has length polynomial in n.

the simply-typed λ-calculus, where the initial λ-term corresponding to a computation has length bounded by a fixed polynomial in n, we may then conclude that the parallel β-reduction *cannot* be unit cost. The reason is simple: were a parallel β-step implemented in unit cost, we would have shown that PTIME equals DTIME$[2^n]$, since an exponential-time computation on an input of size n can be compiled into a short (with length polynomial in n) typed λ-term, and that term normalizes in a polynomial number of parallel β-steps to a Boolean value indicating acceptance or rejection of the input. We would then have simulated an exponential-time computation in polynomial time, and this conclusion contradicts the *time hierarchy theorem* (see, e.g., [HU79]), which asserts that polynomial time is properly contained in exponential time.§

In fact, for any integer $\ell \geq 0$, we can construct generic reductions of this kind for DTIME$[\mathbf{K}_\ell(n)]$. The consequence is that the cost of a sequence of n parallel β-steps is not bounded by any of the *Kalmar-elementary recursive functions* $\mathbf{K}_\ell(n)$. Observe that were the cost contained in $O(\mathbf{K}_\ell(n))$, then by simulating arbitrary computations in DTIME$[\mathbf{K}_{\ell+1}(n)]$, and requiring only $O(n)$ parallel β-steps (via the η-expansion method) at cost $O(\mathbf{K}_\ell(n))$, we would have shown that DTIME$[\mathbf{K}_\ell(n)]$ equals DTIME$[\mathbf{K}_{\ell+1}(n)]$, which is again contradicted by the time hierarchy theorem.

We shall not enter in the complex and tricky details of encoding Turing Machines in the simply typed λ calculus, but we shall content ourselves with providing the main ideas. The interested reader may consult [Mai92, AM97].

Rather than code directly in the simply-typed λ-calculus it is convenient to use an equivalently powerful logical intermediate language: higher-order logic over a finite base type.

Definition 10.1.18 Let $\mathcal{D}_1 = \{\mathbf{true}, \mathbf{false}\}$, and define $\mathcal{D}_{t+1} = powerset(\mathcal{D}_t)$. Let x^t, y^t, z^t be variables allowed to range over the elements of $\mathcal{D}_t$. The *prime formulas* of the logic are x^0, $\mathbf{true} \in y^1$, $\mathbf{false} \in y^1$, and $x^t \in y^{t+1}$. A formula of the logic is any Φ built up out of prime formulas, the usual logical connectives $\vee$, $\wedge$, $\rightarrow$, $\neg$, and the quantifiers $\forall$ and $\exists$.

Meyer [Mey74] proved that the problem of deciding if a generic formula Φ of this logic is true under the usual interpretation requires nonelementary time.

§ Readers familiar with the *diagonalization* technique from the proof of undecidability of the Halting Problem should recognize that in exponential time, one can diagonalize over every polynomial time computation.

The relation of this analysis to typed λ-calculus was originally pointed out by Statman [Sta79] and then revisited and simplified by Mairson [Mai92]. As a matter of fact, the truth of formulas in this logic can be decided by compiling them into short typed λ-terms, and using the power of primitive recursion to realize quantifier elimination and to decide the truth of prime formulas. In particular, primitive recursion is easily implemented using *list iteration*, a straightforward programming technique of functional programming.

The main result is the following [Mai92, AM97]:

Theorem 10.1.19 *A formula Φ in higher-order logic over the finite base type $\mathcal{D}_1 = \{\mathbf{true}, \mathbf{false}\}$ is true if and only if its typed λ-calculus interpretation $\hat{\Phi} : \mathsf{Bool}$ is $\beta\eta$-equivalent to true $\equiv \lambda x : \tau.\lambda y : \tau.x : \mathsf{Bool}$. Moreover, if Φ only quantifies over universes $\mathcal{D}_i$ for $i \leq k$, then $\hat{\Phi}$ has* order† *at most k, and if we also write $|\Phi|$ to denote the length of logic formula Φ, then $|\hat{\Phi}| = O(|\Phi|(2k)!)$.*

The next step is to use the previous logic to express Turing Machine instantaneous descriptions (IDs) of sufficient size, and to compute the transitive closure of the transition relation between such IDs. The logical coding problems that have to be solved are really tricky, in this case (see [Mai92, AM97]). Again we just recall the main results.

Theorem 10.1.20 *Let M be a fixed Turing Machine that accepts or rejects an input x in $\mathbf{K}_\ell(|x|)$ steps. Then there exists a formula Φ_x in higher-order logic such that M accepts x if and only if Φ_x is true. Moreover, Φ_x only quantifies over universes $\mathcal{D}_i$ for $i \leq (\log^* |x|) + \ell + 6$, and $|\Phi_x| = O(|x| \log^* |x|)$.*

Corollary 10.1.21 *Let M be a fixed Turing Machine that accepts or rejects an input x in $\mathbf{K}_\ell(|x|)$ steps. Then there exists a typed λ-term $\hat{\Phi}_x : \mathsf{Bool}$ such that M accepts x if and only if $\hat{\Phi}_x$ reduces to true $\equiv \lambda x : \tau.\lambda y : \tau.x : \mathsf{Bool}$. Moreover, the bound variables in $\hat{\Phi}_x$ have order $O(\log^* |x|)$, and $|\hat{\Phi}_x| = O(|x| \log^{(c)} |x|)$ for any fixed integer $c > 0$.*

† The *order* of a type is a measure of its higher-order functionality:

$$\begin{aligned} \mathrm{order}(o) &= 0 \\ \mathrm{order}(\alpha \to \beta) &= \max\{1 + \mathrm{order}(\alpha), \mathrm{order}(\beta)\}. \end{aligned}$$

Theorem 10.1.22 *There exists a set of* λ*-terms* $E_n : \mathsf{Bool}$ *which normalize in no more than* n *parallel* β*-steps, where the time needed to implement the parallel* β*-steps, on* any *first-class machine model [vEB90] (and in particular for Lamping's algorithm), grows as* $\Omega(\mathbf{K}_\ell(n))$ *for any fixed integer* $\ell \geq 0$*.*

Corollary 10.1.23 (Readback Lemma) *There exists a set of sharing graphs* G_n *where* $[\![G_n]\!]$ *is bounded by a small fixed polynomial in* n*,* G_n *contains no* β*-redexes, the* λ*-term coded by* G_n *has constant size, and the computational work required to read back the represented term grows as* $\Omega(\mathbf{K}_\ell(n))$ *for any fixed integer* $\ell \geq 0$*.*

We remark also that the basic theorem gives bounds on the complexity of cut elimination in multiplicative-exponential linear logic (MELL), and in particular, an understanding of the "linear logic without boxes" formalism in [GAL92b]. In proof nets for linear logic (see, for example, [Laf95]), the *times* and *par* connectives of linear logic play essentially the same role as apply and λ nodes in λ-calculus; the programming synchronization implemented by the *closure* has its counterpart in proof net *boxes*. Just as Lamping's technology can be used to optimally duplicate closures, it can be used to optimally share boxes. Because the simply-typed λ-calculus is happily embedded in multiplicative-exponential logic, we get similar complexity results: small proof nets with polynomially-bounded number of cuts, but a nonelementary number of "structural" steps to resolve proof information coded by sharing nodes.

10.2 Superposition and higher-order sharing

The analysis of the previous section makes it very clear that the parallel β-step is not even *remotely* a unit-cost operation. The workhorse of any optimal reduction engine is not the parallel β-step, but the bookkeeping overhead of sharing redexes. The bookkeeping must remember *which* redexes are being so parsimoniously shared, and *where* parallel β-reduction takes place.

We used Lamping's technology as a calculating device to prove a theorem about all possible implementations of optimal evaluation. However, it is worth asking the following question: what are Lamping's graphs doing to so cleverly encode so much sharing?

In this section, we discuss two essential phenomena of these data structures: that of *superposition*, and that of *higher-order sharing* (these two

notions are essentially due to Mairson). Superposition is a coding trick where graphs for different values – for example *true* and *false*, or different Church numerals – are represented by a single graph. Higher-order sharing is a device where sharing nodes can be used to code other sharing structures, allowing a combinatorial explosion in the size of graphs. Both of these techniques are used to realize the generic simulations of the previous section.

10.2.1 Superposition

Here is a really simple example of superimposed graphs: the coding of *true* and *false*:

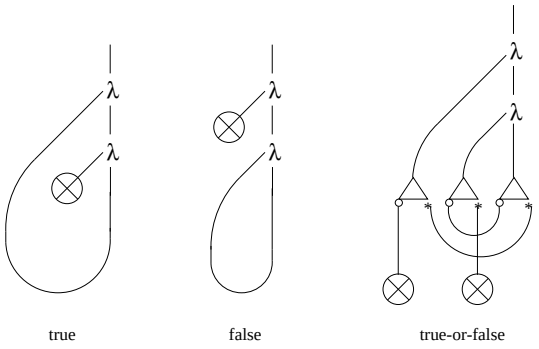


Fig. 10.6. Superposition of *true* and *false*.

Notice how the $\star$-sides of the sharing nodes code *true*, and the $\circ$-sides of the sharing nodes code *false*. The two terms share the λ-nodes that serve as the interface to the external context. A curious consequence of this superposition is that, for example, we can negate *both* Boolean values at the cost usually ascribed to negating only one of them: the negation function merely switches the topological position of two of the sharing nodes.

Such configurations are easily generated by the η-technique. Consider for instance the coding of

$$\lambda \mathsf{c} : \mathsf{Bool} \to \mathsf{o} \to \mathsf{o}.\lambda \mathfrak{n} : \mathsf{o}.\mathsf{c}\ \mathit{true}\ (\mathsf{c}\ \mathit{false}\ \mathfrak{n}) : (\mathsf{Bool} \to \mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}$$

and its η-expanded equivalent, depicted in Figure 10.7.

In the unexpanded term, there is a sharing of c, but not a sharing of the four applications. In the η-expanded term, the coding of (c *true*)

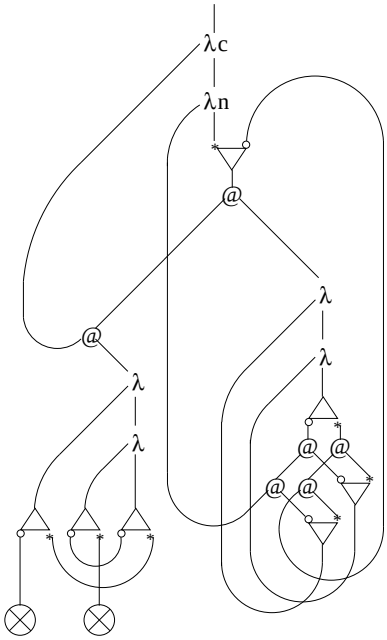


Fig. 10.7. η-expansion of the list of Booleans.

and (c *false*) is shared by a single application, and the argument is exactly our superimposed *true* and *false*. In addition, there is a sharing of ((c *true*) (c *false* n)) and ((c *false*) n). Notice that the companion argument to the superimposed values is just two η-expansions: the ⋆-side, coding ((c *true*) (c *false* n)), leads back into the network of applications (but on the ∘-side), while the ∘-side leads out to the parameter n.

This example explains why SATISFIABILITY can be coded in a trivial number of parallel β-steps. Generalizing the above graph constructions, we may essentially superimposes *every* row of an n-variable truth table into a *single*, shared structure, so that applying a Boolean function to all 2^n rows can be done "simultaneously."

Another beautiful example comes from Church numerals. In Figure 10.8 are two sharing graphs, representing the Church numerals for 2 and 3, in which every application of the "successor" constructor x is shared.

Notice that these examples give us a fairly universal picture of Church numerals, where the λx and λy serve as a uniform interface, the applications are maximally shared, and two numerals are distinguished only by their network of sharing nodes.

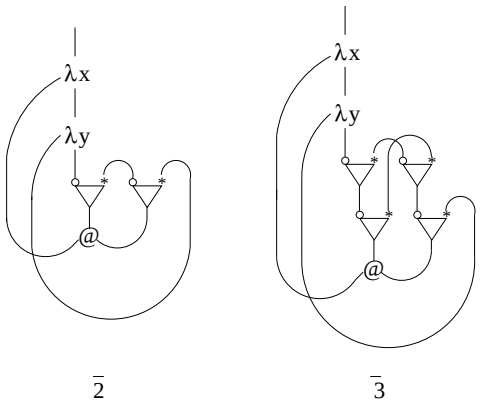


Fig. 10.8. η-expansion of the church numerals 2 and 3.

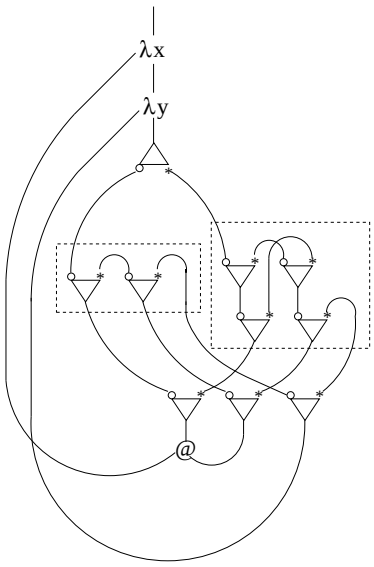


Fig. 10.9. Superimposed 2 and 3.

Now we can code a superimposed representation of many Church numerals, by inserting more sharing nodes to serve as multiplexors and demultiplexors leading to the correct sharing network (Figure 10.10).

This style of computation looks like a λ-calculus optimal evaluation implementation of SIMD (single instruction, multiple data) parallelism. Imagine a list $\lambda c.\lambda n.c\ x_1\ (c\ x_2\ \cdots\ (c\ x_\ell\ n)\cdots)$ where the x_i are Church

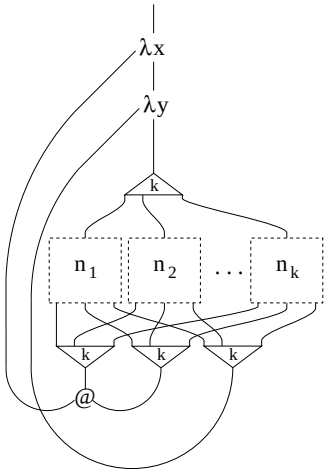


Fig. 10.10. Superimposing Church numerals.

numerals, which gets η-expanded so that the applications $(\mathsf{c}\ x_i)$ are all shared, via superimposed representations of the numerals. We can then, for example, apply any integer function to each of the numerals "simultaneously," that is, counting parallel β-steps, but not counting sharing interactions. The sharing network for each numeral is like a separate processor, serving as multiple data; the λ- and apply-nodes serve as interface to the external context; the multiplexors and demultiplexors replicate and pump the single-instruction stream (the code for, say, factorial) to each of the processors. The real work of the computation becomes *communication*, performing the η-expansion on the lists, and communicating the function to to different processors. But the actual computation associated with the function occurs via interaction of sharing nodes only, our "parallel computation."

10.2.2 Higher-order sharing

Superposition is only one component of the graph reduction technology that supports nonelementary computation in a trivial number of parallel β-steps. The other essential phenomenon is *higher-order sharing*, used to construct enormous networks of sharing nodes.

10.2.2.1 A simple, exponential construction

Let us start with a simple example, used to simulate an exponential number of function applications. As we already remarked, the church numeral two can be described by the graph in Figure 10.11.

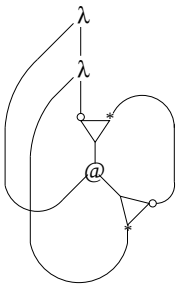


Fig. 10.11. A representation of Church's integer 2

Let us now consider the application of two' to itself. Recall that the application $(\mathfrak{n}\ \mathfrak{m})$ of two church integers $\mathfrak{n}$ and $\mathfrak{m}$ gives the church integer $\mathfrak{m}^{\mathfrak{n}}$, so the expected result is (a representation of) the church integer four. The reduction is shown in Figure 10.12.

The two first reduction steps are β-redex. After these reductions we are left with the term in Figure 10.12.(3), where the subterm $\lambda y.(x(x\,y))$ is shared by means of the two copies of the fan marked "a". Now, this subterm is fully duplicated. This process requires: 2 steps for duplicating the λ and the application; 2 steps for duplicating the fans; 3 steps for effacing all residuals of fans marked with "a". After these 7 steps we are left with the graph in Figure 10.12.(4), where a new β-redex has been put in evidence. Firing this redex, we obtain the final configuration in Figure 10.12.(5). Summing up, we executed 3 β-reductions, and 7 fan-interactions. Note moreover that the final configuration has the same shape of the initial one.

Let us now generalize the previous example. As should be clear, the church integer $2^{\mathfrak{n}}$ can be represented by the graph in Figure 10.13, where we have exactly a sequence of fan-in of length $\mathfrak{n}$ and a corresponding sequence of fan-out of the same length.

Let us now apply this term to itself. By firing the two outermost β-redexes we get the term in Figure 10.14.

As in the case of two, the portion of graph inside the two fans marked with "a" is now fully duplicated. This duplication requires: 2 steps for

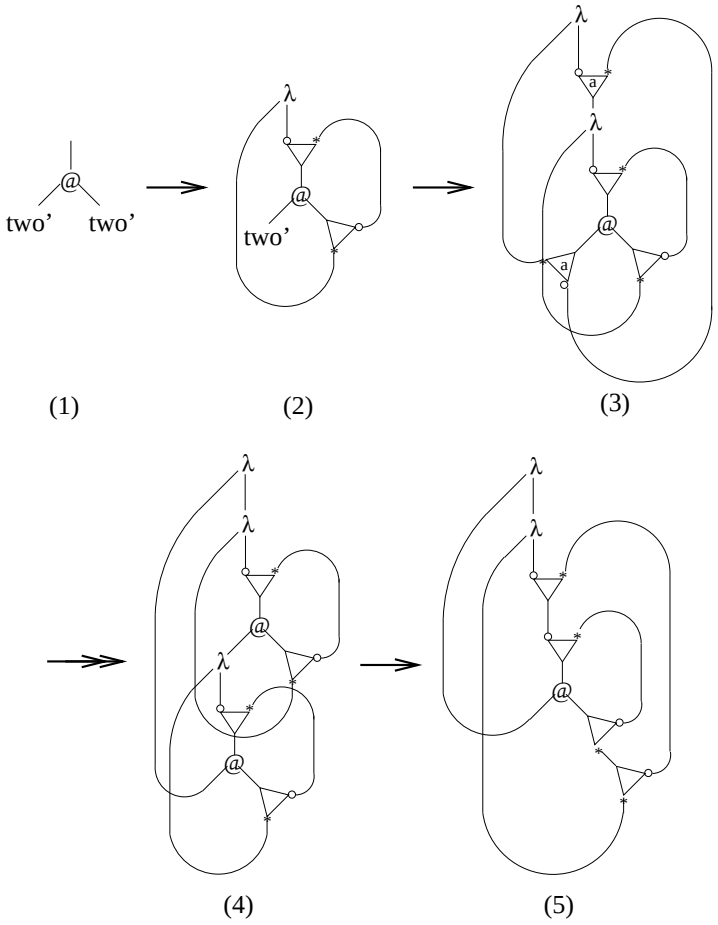


Fig. 10.12. the reduction of (two' two')

duplicating the λ and the application; 2n steps for duplicating fans; n+2 steps for effacing fans. This gives a total of 4 + 3n operations.

After these reductions, we get the configuration in Figure 10.15.

A new β-redex has been created. By firing this redex we obtain the graph in Figure 10.16. Now, this graph has the same shape of the graph in Figure 10.14, and we can iterate our reasoning.

In particular, the duplication of the innermost part of the graph will now require 4 + 3*(2n) operations. Then, we shall have a new β-redex, and by its firing, we shall get a graph of the same shape of Figure 10.16 but where the innermost sequences of fans have length 4n (this length

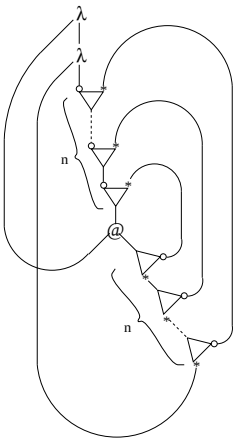


Fig. 10.13. a representation of 2^n

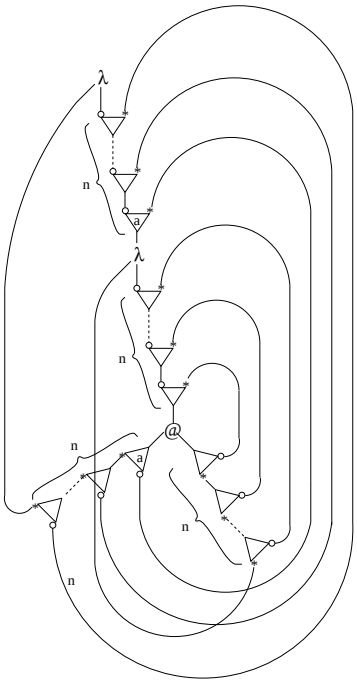


Fig. 10.14. the reduction of $(2^n\ 2^n)$

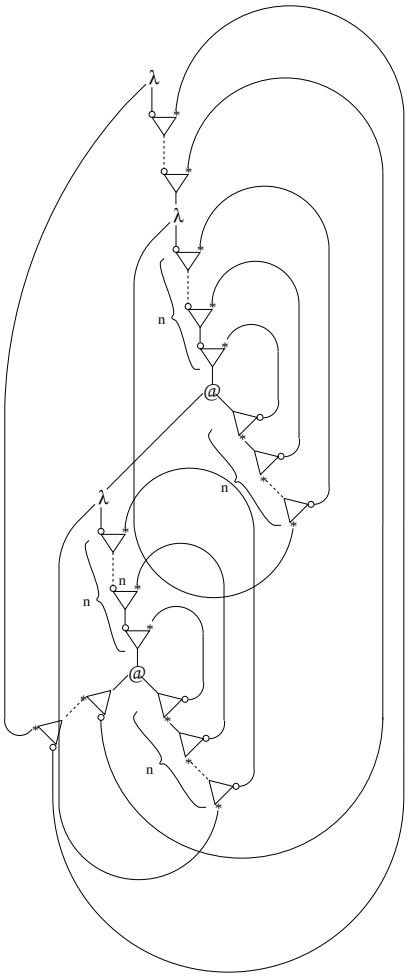


Fig. 10.15. the reduction of $(2^n\ 2^n)$

is doubled at every iteration of the process), while the length of the outermost sequences is decremented by one.

Summing up, the total number of fan-interactions is given by

$$((4+3n)+(4+3*(2n))+(4+3*(4n))+\ldots+(4+3*(2^{n-1}n)))$$

$$=4n+3n*\sum_{i=0}^{n-1}2^i=4n+3n*(2^n-1)=n*(3*2^n+1)$$

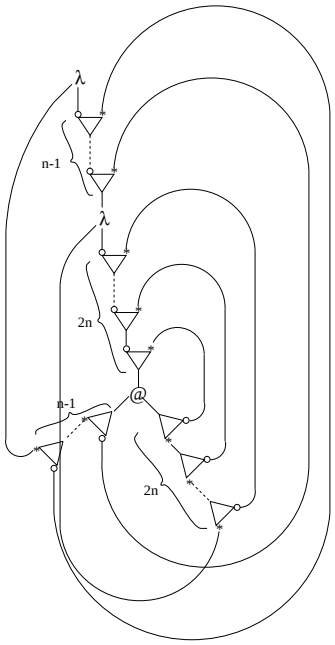


Fig. 10.16. the reduction of $(2^n\ 2^n)$

In contrast, we have executed just n β-reductions in the main loop, plus two at the very beginning, for a total of $n+2$ family reductions.

The previous construction is essentially obtained in the reduction of

$$g = \lambda n.(n\ \delta\ two'\ I\ q)$$

where $\delta = \lambda x.(xx)$, $two' = \lambda x.\lambda y.(\lambda z.(z(z\ y))\ \lambda w.(x\ w))$, I is the identity and q is some constant. If n is a church integer, $(g\ n)$ obviously reduces to q. As a function of n, the term $(n\ \delta\ two')$ corresponds to the church integer a_n, in the succession $a_0 = 2; a_{i+1} = a_i^{a_i}$.

It is easy to prove that the number $f(n)$ of family reductions grows as

$$f(n) = 9 + 3 * n + \sum_{i=0}^{n-1} b_i$$

where $b_i = log(a_i)^{\dagger}$.

Finally, let us consider the number of fan-interactions. For each application of $(\delta\ a_i)$, we have $1 + 4 + 3 * b_i$ interactions for duplicating

† b_n can be equivalently defined as $b_0 = 1; b_{i+1} = b_i * 2^{b_i}$.

a_i, plus $b_i + 3 * b_{i+1}$ operations in the reduction of $(a_i\ a_i)$ (recall that $b_{i+1} = b_i * 2^{b_i}$). Moreover, we have one single operation internal to two', $5 * (n-1)$ operations for creating all copies of δ, and b_n final operations of fan-effacement when we apply the external parameters.

Summing up, the number $c(n)$ of fan-interactions (for $n > 0$) is given by the formula:

$$c(n) = 10 * n - 4 + 4 * \sum_{i=0}^{n-1} b_i + 3 * \sum_{i=1}^{n} b_i + b_n$$

$$= 10 * n + 7 * \sum_{i=1}^{n-1} b_i + 4 * b_n > 4 * b_n$$

Note now that $2^{\sum_{i=0}^{n-1} b_i} = b_n$. So,

$$2^{f(n)} = 2^{9+3n+\sum_{i=0}^{n-1} b_i} = 2^{9+3n} * b_n$$

For $n \geq 3$, it is easy to show that $2^{9+3n} < 2^8 * b_n$. Hence:

$$2^{f(n)} < 2^8 * b_n^2 < 2^4 * c(n)^2.$$

and finally (for any $n \geq 3$)

$$c(n) > \frac{1}{4} * 2^{\frac{f(n)}{2}}$$

We can also easily prove that, for any n, $c(n) \leq 2^{f(n)}$. On one side we have $2^{f(n)} > 2^9 * b_n$. On the other side, $\sum_{i=0}^{n-1} b_i \leq b_n$ and obviously $n \leq b_n$, so

$$c(n) < 21 * b_n < 2^9 * b_n < 2^{f(n)}$$

The previous formulas have been also experimentally confirmed by our prototype implementation of (a variant of) Lamping's algorithm: the Bologna Optimal Higher-Order Machine (see Chapter 12). The results of the computation of the function g are shown in Figure 10.17.

The four columns in the table are, respectively, the user time required by the computation (on a Sparc-station 5), the total number of interactions (comprising the "oracle"), the length of the family reduction (app-lambda interactions), and the total number of fan-interactions.

It is also very easy to find examples of exponential explosion with respect to a *linear* grow of the number of family reductions. A simple example is provided by the λ-term $h = \lambda n.(n\ two'\ two'\ I\ q)$. In this case the number of family reductions $f(n)$ grows linearly in its input n

Input	user	tot. inter.	fam.	fan-inter.
(g zero)	0.00 s.	38	9	2
(g one)	0.00 s.	64	13	18
(g two)	0.00 s.	200	18	66
(g three)	15.90 s.	2642966	29	8292

Fig. 10.17. The function g

(in particular, $f(n) = 9 + 3 * n$), while the number of fan-interactions $c(n)$ is given by the formula

$$c(n) = 12 * n - 2 + 4 * 2^n$$

In particular, for any n,

$$2^{\frac{f(n)-7}{3}} \leq c(n) \leq 2^{f(n)}$$

In the table of Figure 10.18, you will find the experimental results in BOHM. In this case (the term is typable), we also make a comparison with two standard (respectively, strict and lazy) implementations such as Caml-Light and Yale Haskell.

CamlLight [LM92] is a bytecoded, portable implementation of a dialect of the ML language (about 100K for the runtime system, and another 100K of bytecode for the compiler, versions for the Macintosh and IBM PC are also available) developed at the INRIA-Rocquencourt (France). In spite of its limited dimensions, the performance of CamlLight is quite good for a bytecoded implementation: about five times slower than SML-NJ. We used CamlLight v. 0.5.

Yale Haskell [The94] is a complete implementation of the Haskell language developed at Yale University. The Haskell compiler is written in a small Lisp dialect similar to Scheme which runs on top of Common Lisp. Haskell programs are translated into Lisp, and then compiled by the underlying Lisp Compiler. We used Yale Haskell Y2.3b-v2 built on CMU Common Lisp 17f.

The results of the test should give a gist of the power of the optimal graph reduction technique.

10.2.2.2 An elementary construction

The key idea in the previous construction, and in more sophisticated examples of higher-order sharing, is the pairing of sharing nodes with

Input	BOHM				Caml-Light	Haskell
	user	tot. inter.	fam.	fan-inter.	user	user
(h one)	0.00 s.	67	12	18	0.00 s.	0.00 s.
(h two)	0.00 s.	119	15	38	0.00 s.	0.02 s.
(h three)	0.00 s.	204	18	66	0.00 s.	0.18 s.
(h four)	0.00 s.	414	21	110	1.02 s.	51.04 s.
(h five)	0.00 s.	1054	24	186	??	??
(h six)	0.02 s.	3274	27	326		
(h seven)	0.07 s.	11534	30	594		
(h eight)	0.26 s.	43394	33	1118		
(h nine)	1.01 s.	168534	36	2154		
(h ten)	4.04 s.	664554	39	4214		

Fig. 10.18. The function h

wires from $\circ$ to $\star$-sides, so that we can enter the same graph in two different "modes." (see Figure 10.19).

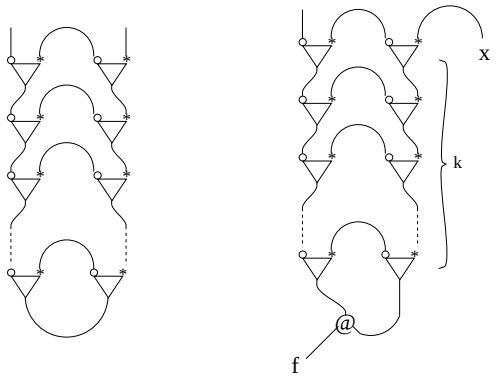


Fig. 10.19. Exponential path and exponential function application

In this case, a naive version of context information can be constructed as mere stacks of $\star$ and $\circ$. When a stack enters a sharing node at its interaction port, the top token on the stack is popped, and used to

determine whether the path to follow is along the $\circ$ or $\star$-auxiliary ports of the node (and conversely, when entering a sharing node).

However, this construction merely shares an application node; to get a truly powerful network of nodes, we need to be able to share *sharing nodes* as well. For example, consider the network of Figure 10.20.

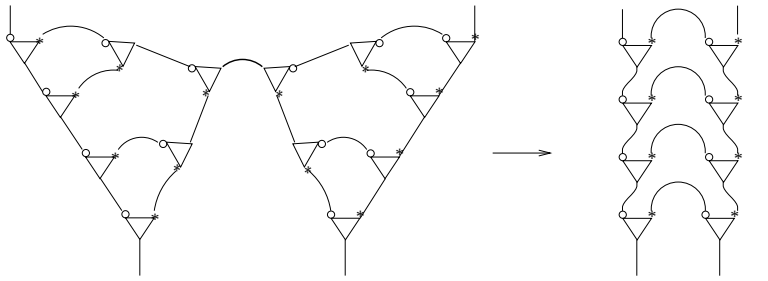


Fig. 10.20. A simple sharing network.

By linking two symmetric copies of this construction via the marked wire, we get a "stack" of sharing nodes implementing the standard exponential construction. Iterating the idea of the above figure, we can derive a doubly-exponential stack, as in Figure 10.21.

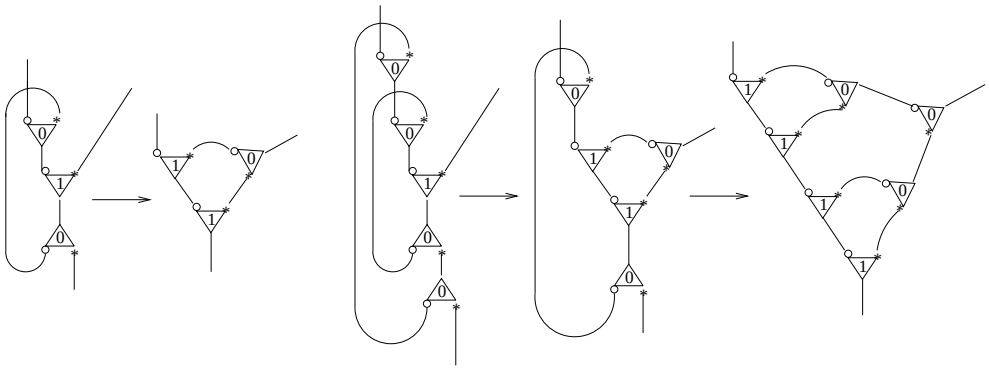


Fig. 10.21. A doubly-exponential stack of sharing nodes.

Now we go one step further, considering the *nested* construction of Figure 10.22, which hints at how a network of $\mathfrak{n}$ sharing nodes can expand to a network of size $\mathbf{K}_\ell(\mathfrak{n})$:

After these illustrative examples, we can describe the basic idea of the *elementary construction* (in the sense of Kalmar). We define a sequence

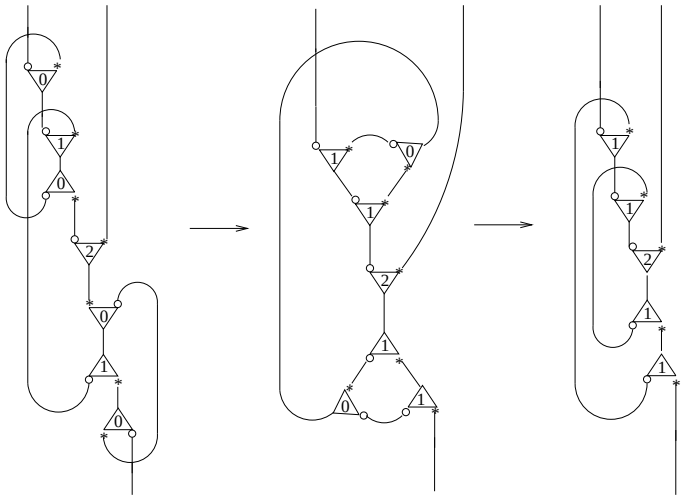


Fig. 10.22. A "nested" construction.

of *networks* N_j, where N_j contains sharing nodes at *levels* (not to be confused with *indices*) $0 \leq \ell \leq i$. Like a sharing node, every network will have one principal port, two auxiliary ports (identified as the $\circ$ and $\star$-ports), and a distinguished sharing node that we will call the *core* node. Given a sharing network N, we will write $\overline{N}$ to mean N with black and white exchanged on the auxiliary ports of the core node of N.

The network N_0 is a single sharing node at level 0, which is by process of elimination the core node. To construct N_{j+1}, we combine N_j, $\overline{N_j}$, and a new core node at level $j+1$, attaching the principal ports of the core node and $\overline{N_j}$, the principal port of N_j to the $\circ$-auxiliary port of the core node, and the $\star$-auxiliary node of N_j to the $\circ$-auxiliary node of $\overline{N_j}$ (see Figure 10.23).

The principal port of N_{j+1} is the black external port of $\overline{N_j}$; auxiliary ports are the $\star$-auxiliary port of the core node, and the $\circ$-auxiliary port of N_j. Note that the $\circ$ and $\star$-ports of N_{j+1} and $\overline{N_{j+1}}$ are essentially "inherited" by the core node.

It should be clear that N_j has $2^j - 1$ sharing nodes. We define a naive oracle for interactions between sharing nodes: nodes at identical level annihilate, otherwise they duplicate.

Lemma 10.2.1 *The network* N_j *(respectively,* $\overline{N_j}$*) normalizes to a complete binary tree with* $\mathbf{K}_j(1)$ *leaves. The leaves are connected to sharing*

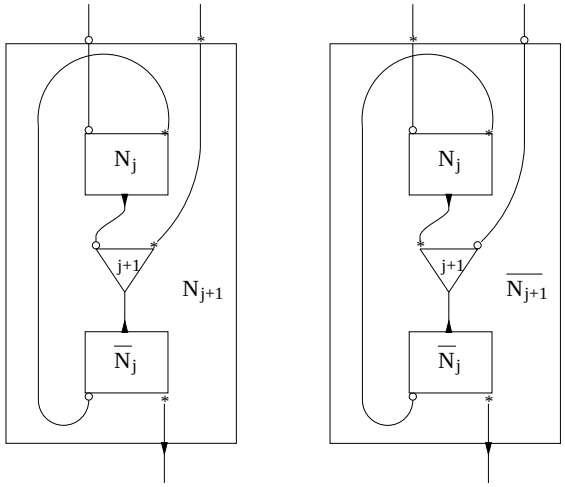


Fig. 10.23. The generic construction of an elementary network.

nodes at level $j-1$ *on their black (respectively, white) auxiliary ports; the remaining auxiliary port is connected to the primary port of the node at the adjacent leaf, as in Figure 10.21.*

Proof By induction. The lemma is trivially true for $j=0$. For $j=i+1$, use the induction hypothesis to normalize N_i and $\overline{N_i}$, producing the network in Figure 10.20. The two binary trees now annihilate each other, and the two stacks of $\mathbf{K}_i(1)$ sharing nodes then create a complete binary tree with $\mathbf{K}_{i+1}$ copies of the core node, with a complete binary tree linked to these copies at the leaves. □

This kind of sharing network results from the parallel β-reduction of the η-expanded term $\mathbf{2}_j \equiv \overline{2}\,\overline{2}\,\cdots\,\overline{2}$ of j $\overline{2}$s, where $\overline{2}$ is the Church numeral for 2. This term has length $O(2^j)$ because of explicit type information that doubles for each additional $\overline{2}$; $\mathbf{2}_j$ normalizes in $O(j)$ parallel β-steps to the Church numeral for $\mathbf{K}_j(1)$. The exponential length is sufficient to code a copy of N_j and $\overline{N_j}$; after normalization, these networks expand to construct $\mathbf{K}_j(1)$ function applications. The same computational phenomenon is evident in the coding of the iterated powerset, though not as straightforward to describe.

10.3 Conclusions

We have seen that the cost of implementing a sequence of n parallel β-reduction steps in the reduction of a λ-term cannot be bounded in general by any Kalmar-elementary function $\mathbf{K}_\ell(n)$. Given that the parallel β-step is one of the key ideas in optimal reduction, it makes sense to consider whether the idea of optimal evaluation has any implementative relevance, and in particular if the complex graph reduction technology that is the main subject of this book is of any good, in practice.

The answer is yes, of course. The previous result merely gives a lower bound in a particularly "bad" case: the simulation of a generic computation, which is "incompressible" by the time hierarchy theorem. But the real question are:

(i) With respect to more traditional implementations, does Lamping's graph reduction algorithm introduce any major overhead, in general?
(ii) How much can we gain in "good" cases?

As for the second question, we have a lot of examples where optimal graph reduction can drastically reduce the actual cost of the computation (from exponential to linear, say, but maybe more: we do not know yet).

The first question looks more delicate. Clearly (and the proof of our complexity results should be enough to convince the reader) optimal graph reduction is really parsimonious in its duplication strategy. This insight and intuition can be formalized more precisely by the fact that the number of interactions of λ nodes, apply nodes, and sharing nodes in optimal graph reduction is bounded by a polynomial in the number of *unique* Levy's labels generated in a reduction [LM97]. If we believe that a label identifies a set of similar subexpressions, this result means that the graph reduction algorithm is really maximizing sharing up to a polynomial factor.

It remains, however, to give a definitive solution to the problem of accumulation of *control operators* discussed in the previous chapter, and the equally annoying problem of garbage collection.

11
Functional Programming

So far, we have been merely dealing with λ-calculus. Facing the problem of using Lamping's technique for an actual implementation of a Functional Programming Language, we must obviously extend this technique to cover a rich set of primitive data types, as well as other computational constructs, such as conditionals, recursion and so on. Luckily, there is a simple and uniform way for taking into account all these aspects of a real programming language, starting from the relation between optimal reduction and Linear Logic.

The main idea behind Linear Logic is to make a clean distinction between the *logical* part of the calculus (dealing with the logical connectives) and the *structural* part (dealing with the management of hypotheses, which can be considered as physical resources). These two different aspects of logical calculi have their obvious correspondent (via the Curry-Howard analogy) inside λ-calculus: on one side, we have the *linear* part (the linear λ-calculus), defining the syntactical operators (the arity, the "binding power", etc.) and their interaction (β-reduction); on the other side, we have the *structural* part, taking care of the management (sharing) of resources (i.e., of subexpressions). Summarizing in a formula:

$$\lambda\text{-calculus} = \text{linear } \lambda\text{-calculus} + \text{sharing}.$$

By the previous chapters, it is clear that Lamping's sharing operators (fan, croissant and square bracket) provide a very abstract framework for an (optimal) implementation of the structural part, which is then interfaced to the linear (or logical) part of the calculus. Therefore, it should be clear that Lamping's technique *smoothly generalizes* to a larger class of calculi, just replacing linear λ-calculus with any arbitrary linear calculus that preserves a smooth interface between the linear and the

structural subparts of the system. By the way, the critical point of this generalization is the interface between the two parts of the system. Let us be more precise on this last point. The natural correpondence between structural operators of sharing graphs and structural rules of Linear Logic strongly suggests that the system extending or replacing λ-calculus must be interpretable in some extension of the intuitionistic part of Linear Logic. In other words, the chosed linear calculus must preserve the *strong logical foundation* of the system.

Luckily, in literature there is a natural—and up to our knowledge unique—candidate for the linear calculus replacing λ-calculus: Lafont's Interaction Nets [Laf90]. Therefore, dropping the linearity constraint in Interaction Nets, we eventually obtain our candidate calculus, introduced in [AL94b] under the name of *Interaction Systems*. In the spirit of the equation above, we can then summarize the result by the formula:

$$\text{Interaction Systems} = \text{Interaction Nets} + \text{sharing}.$$

The main (logical) properties of Interaction Systems (borrowed from Interaction Nets) are the following:

- the syntactical operators are classified in two dual groups: *constructors* and *destructors*;
- each operator has a distinguished port, called its main or *principal port*, where it can interact with a dual operator;
- each reduction is an interaction between a *constructor-destructor pair* (thas is, a logical cut).

From the point of view of optimal reduction, the relevance of having a distinguished principal port for each operator should be clear. The deep reason because of which Lamping's technique is optimal in the case of λ-calculus is that duplication of applications and λ-nodes is only allowed at their principal port. Therefore, this property must be preserved when extending the system by adding new operators to the signature—this is what we meant saying that the extension must preserve a clean interface between the logical and the structural part.

The logical nature of Interaction Systems (IS's) becomes particularly clear comparing them with other Higher-order rewriting systems, such as Klop's Combinatory Reduction Systems (CRS's) [Klo80]. As a matter of fact, Interaction Systems are just the subclass of Combinatory Reduction Systems where the Curry-Howard analogy *still makes sense*. This means that we can associate to every IS a suitable logical (intuitionistic) system: constructors and destructors respectively correspond

to right and left introduction rules, interaction is cut, and computation is cut-elimination. As a consequence, Interaction Systems have a nice *locally sequential nature*: in particular, the leftmost outermost reduction strategy reduces needed redexes only.

11.1 Interaction Systems

An Interaction System is defined by a *signature* Σ and a set of *rewriting rules* R. The signature Σ consists of a denumerable set of *variables* and a set of *forms*.

Variables will be ranged over by $x, y, z, \cdots$, possibly indexed. Vectors of variables will be denoted by $\vec{x}_i$ where i is the length of the vector (often omitted).

Forms are partitioned into two disjoint sets Σ^+ and Σ^-, representing *constructors* (ranged over by $\mathbf{c}$) and *destructors* (ranged over by $\mathbf{d}$). Each form of the syntax can be a binder. In the arity of the form we must specify not only the number of arguments, but also, for each argument, the number of bound variables. Thus, the *arity* of a form $\mathbf{f}$, is a finite (possibly empty) sequence of naturals. Moreover, we have the constraint that the arity of every destructor $\mathbf{d} \in \Sigma^-$ has a leading 0 (i.e., it cannot bind over its first argument). The reason for this restriction is that, in Lafont's notation [Laf90], at the first argument we find the *principal port* of the destructor, that is the (unique) port where we will have interaction (local sequentiality).

Expressions, ranged over by $t, t_1, \cdots$, are inductively generated as follows:

(i) every variable is an expression;

(ii) if $\mathbf{f}$ is a form of arity $k_1 \cdots k_n$ and $t_1, \cdots, t_n$ are expressions then

$$\mathbf{f}(\vec{x}^1_{k_1}.\, t_1, \cdots, \vec{x}^n_{k_n}.\, t_n)$$

is an expression.

Free and bound occurrences of variables are defined in the obvious way. As usual, we will identify terms up to renaming of bound variables (α-conversion).

Rewriting rules are described by using schemas or metaexpressinos. A *metaexpression* is an expression with *metavariables*, ranged over by $X, Y, \cdots$, possibly indexed (see [Acz78] for more details). Metaexpressions will be denoted by $H, H_1 \cdots$.

A *rewriting rule* is a pair of metaexpressions, written $H_1 \rightarrowtail H_2$, where

H_1 (the *left hand side* of the rule or lhs for short) has the following format:

$$\mathbf{d}(\mathbf{c}(\vec{x}^1_{k_1}.X_1, \cdots, \vec{x}^m_{k_m}.X_m), \cdots, \vec{x}^n_{k_n}.X_n)$$

where $i \neq j$ implies $X_i \neq X_j$ (i.e., *left linearity*). The arity of $\mathbf{d}$ is $0k_{m+1} \cdots k_n$ and that of $\mathbf{c}$ is $k_1 \cdots k_m$.

The *right hand side* H_2 (rhs, for short) is every *closed* metaexpression, whose metavariables are already in the lhs of the rule, built up according to the following syntax:

$$H \ ::= \ x \ \mid \ \mathbf{f}(\vec{x}^1_{a_1}.H_1, \cdots, \vec{x}^j_{a_j}.H_j) \ \mid \ X_i[^{H_1}/_{x^i_1}, \cdots, ^{H_{k_i}}/_{x^i_{k_i}}]$$

where the expression $X[^{H_1}/_{x_1}, \cdots, ^{H_n}/_{x_n}]$ denotes a meta-operation of substitution (as the one of λ-calculus) defined in the obvious way.

Finally, the set of rewriting rules must be *non-ambiguous*, i.e., there exists at most one rewriting rule for each pair **d**-**c**.
Interaction Systems are a subsystem of Klop's (Orthogonal) Combinatory Reduction Systems [Klo80, Acz78]. We just added a bipartition of operators into constructors and destructors, and imposed a suitable constraint on the shape of the lhs of each rule. As a subclass of non ambiguous, left-linear CRS's, Interaction Systems inherit all good properties of the former ones (Church Rosser, finite development, ...).

Example 11.1.1 (λ-calculus) The application @ is a destructor of arity 00, and λ is a constructor of arity 1. The only rewriting rule is β-reduction:

$$@(\lambda(x.X), Y) \ \rightarrow \ X[^Y/_x].$$

Example 11.1.2 (Booleans) Booleans are defined by two constructors **T** and **F** of arity ε (two constants). Then you may add your favorite destructors. For instance the logical conjunction is a destructor **and** of arity 00, with the following rewriting rules:

$$\mathbf{and}(\mathbf{T}, X) \rightarrow X \qquad\qquad \mathbf{and}(\mathbf{F}, X) \rightarrow \mathbf{F}$$

Another typical destructor is the *if-then-else* operator **ite**, of arity 000:

$$\mathbf{ite}(\mathbf{T}, X, Y) \rightarrow X \qquad\qquad \mathbf{ite}(\mathbf{F}, X, Y) \rightarrow Y$$

Example 11.1.3 (Lists) Lists are defined by two constructors **nil** and

cons of arity ε and 00, respectively. The typical destructors **hd**, **tl** and **isnil** (all of arity 0) may be defined by the following rules:

$$\mathbf{hd}(\mathbf{cons}(X,Y)) \to X \qquad \mathbf{tl}(\mathbf{cons}(X,Y)) \to Y$$
$$\mathbf{isnil}(\mathbf{nil}) \to \mathsf{T} \qquad \mathbf{isnil}((\mathbf{cons}(X,Y)) \to \mathsf{F}$$

Example 11.1.4 (Primitive Recursion) The schemas for *Primitive Recursion* does perfectly fit inside Interaction Systems. In this case, we have only two constructors `0` and `succ`. Any new function `d` defined by primitive recursion corresponds to a new destructor **d** with interaction rules of the following kind:

$$\mathbf{d}(\mathtt{0}, X) \to \mathbf{h}(X)$$
$$\mathbf{d}(\mathtt{succ}(X), Y) \to \mathbf{f}(X, Y, \mathbf{g}(X,Y))$$

For instance, we may define sum and product by the following IS-rules:

$$\mathtt{add}(\mathtt{0}, X) \to X$$
$$\mathtt{add}(\mathtt{succ}(X), Y) \to \mathtt{succ}(\mathtt{add}(X, Y))$$

$$\mathtt{mult}(\mathtt{0}, X) \to \mathtt{0}$$
$$\mathtt{mult}(\mathtt{succ}(X), Y) \to \mathtt{add}(Y, \mathtt{mult}(X, Y))$$

Example 11.1.5 (Integers) For practical purposes, we cannot obviously use the unary representation of integers given by `0` and `succ`. The obvious solution is to consider each integer `n` as a distinguished constructor **n**. Then, we may define arithmetical operations in constant time. The only problem is the strictly sequential nature of IS's, that imposes interaction on a distinguished port of the form. For instance, we may define

$$\mathbf{add}(\mathbf{m}, X) \to \mathbf{add_m}(X) \qquad \mathbf{add_m}(\mathbf{n}) \to \mathbf{k}$$

where $\mathtt{k} = \mathtt{n+m}$. Note that we have an infinite number of forms, and also an infinite number of rewriting rules.

Example 11.1.6 (General Recursion) The definition of the recursion operator

$$\mu x.M \to M[{}^{\mu x.M}/_{x}]$$

is less straightforward than the previous examples, for in this case we do not have a pair of interacting forms. The obvious idea is to introduce a suitable "dummy" operator interacting with μ. Then, according if we

choose to introduce a constructor or a destructor, we eventually get the two following encodings:

$$\begin{aligned} \mathbf{d}_\mu(\mu(x.X)) &\rightarrow X[^{\mathbf{d}_\mu(\mu(x.X))}/_x] \\ \mu(\mathbf{c}_\mu, x.X) &\rightarrow X[^{\mu(\mathbf{c}_\mu, x.X)}/_x] \end{aligned}$$

Consequently, in the first case μ is a constructor, while in the second it is a destructor. In other words, and more precisely, μ is a sort a constructor-destructor pair, permanently interacting with itself.

11.1.1 The Intuitionistic Nature of Interaction Systems

In this section we shall recall the logical, intuitionistic nature of Interaction Systems, and their relations with Lafont's Interaction Nets. The aim is to introduce and clarify some essential notions of IS-forms (polarities, principal and auxiliary ports, bound ports, inputs and outputs, etc.).

11.1.1.1 Statics

An Intuitionistic System, in a *sequent calculus* presentation (*à la* Gentzen) (see [Gir89b]), consists of expressions, named *sequents*, whose shape is $A_1, \cdots A_n \vdash B$ where A_i and B are formulas. Inference rules are partitioned into three groups (in order to emphasize the relationships with IS's, we write rules by assigning terms to proofs):

Structural Rules

$$(\textit{Exchange}) \quad \frac{, x : A, y : B, \Delta \vdash t : C}{, y : B, x : A, \Delta \vdash t : C}$$

$$(\textit{Contr.}) \quad \frac{, x : A, y : A \vdash t : C}{, z : A, \Delta \vdash t[^z/_x, ^z/_y] : C} \qquad (\textit{Weak.}) \quad \frac{\vdash t : C}{, z : A \vdash t : C}$$

Identity Group

$$(\textit{Identity}) \quad \frac{}{x : A \vdash x : A} \qquad (\textit{Cut}) \quad \frac{\vdash t : A \qquad \Delta, x : A \vdash t' : B}{, \Delta \vdash t'[^t/_x] : B}$$

Logical Rules

These are the "peculiar" operations of the systems, for they allow to introduce new formulae (i.e., new types) in the proof. The unique new

formula P introduced by each logical rule is called the *principal* formula of the inference. The inference rule is called *right*, when the principal formula is in the rhs of the final sequent, and *left* otherwise. Right and left introduction rules respectively correspond with *constructors* and *destructors* in IS's. The shape of these rules is:

$$\frac{\Gamma_1, \vec{x}^1 : \vec{A}^1 \vdash t_1 : B_1 \quad \cdots \quad \Gamma_n, \vec{x}^n : \vec{A}^n \vdash t_n : B_n}{\Gamma_1, \cdots, \Gamma_n \vdash \mathbf{c}(\vec{x}^1.t_1, \cdots, \vec{x}^n.t_n) : P}$$

for right introduction rules (constructors), and

$$\frac{\Gamma_1, \vec{x}^1 : \vec{A}^1 \vdash t_1 : B_1 \quad \cdots \quad \Gamma_m, \vec{x}^m : \vec{A}^m \vdash t_m : B_m \qquad \Delta, z : C \vdash t : D}{\Gamma_1, \cdots, \Gamma_m, \Delta, y : P \vdash t[^{\mathbf{d}(y, \vec{x}^1.t_1, \cdots, \vec{x}^m.t_m)}/_z] : D}$$

for left introduction rules (destructors). The contexts Γ_i are pairwise different.

A canonical example is logical implication, that gives the expressions of typed λ-calculus:

$$(\to l)\ \frac{\Delta \vdash t : A \qquad z : B, \quad \vdash t' : C}{\Delta, y : A \to B, \quad \vdash t'[^{@(y,t)}/_z] : C} \qquad (\to r)\ \frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda(x.t) : A \to B}$$

An immediate consequence of the above construction is that every proof of an Intuitionistic System may be described by an IS-expression. In particular, following Lafont, we may provide a graphical representation of IS-forms. This will explain some important relations between the arity of the forms and the way in which they link upper sequents in a proof.

In the notation of Interaction Nets, a proof of a sequent $A_1, \ldots, A_n \vdash B$ is represented as a graph with $n+1$ conclusions, n with a negative type (the inputs) and one with a positive type (the output). In particular, an axiom is just a line with one input and one output.

Every logical rule is represented by introducing a new operator in the net (a new labeled node in the graph). The operator has a *principal (or main) port*, individuated by an arrow, that is associated with the *principal formula* of the logical inference. For instance, the two logical rules for implications are illustrated in Figure 11.1.

The principal port of each operator may be either an input or an output. In the first case it corresponds to a new logical assumption in the left hand side of the sequent (as for @), and the operator is a *destructor*; in the second case it corresponds to the right hand side of

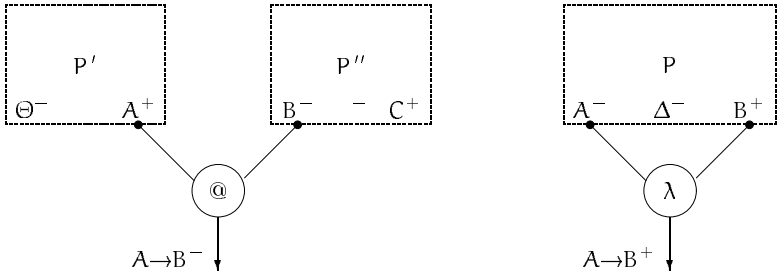


Fig. 11.1. The graphical representation of @ and λ.

the sequent (as for λ), and the operator is a *constructor*. The other ports of the agents are associated with the *auxiliary* formulae of the inference rule, that is the distinguished occurrences of formulae in the upper sequents of the rule. In the two rules above, the auxiliary formulae are A and B.

The auxiliary ports of an agent may be either inputs or outputs, independently from the fact that it is a constructor or a destructor. Actually, in the general theory of Interaction Nets, which is inspired by *classical* (linear) logic, there is a complete symmetry between constructors and destructors, and no restriction at all is imposed on the type of the auxiliary ports (in other words, there is a complete symmetry between inputs and outputs). On the contrary, the fact of limiting ourselves to the intuitionistic case imposes some obvious "functional" constraints.

Note first that auxiliary formulae may come from different upper sequents or from a single one. For instance, the auxiliary ports of @ are in different upper sequents, while those of λ are in a same one. Lafont expresses this fact by defining *partitions* of auxiliary ports. So, in the case of @, A and B are in different partitions, while in the case of λ they are in the same partition. Note that the concept of partition is obviously related to that of binding. In particular, a partition is a singleton if and only if the arity of the form for that port is 0. Moreover, the polarity of an auxiliary port is the opposite of the polarity of the conclusion it has to be matched against. Then, the intuitionistic framework imposes the following constraints:

- In every partition there is at most one negative port. If a negative port exists, we shall call it an *input* partition; otherwise it is an *output*. The positive ports of an input partition are called *bound ports* of the form (they are the ports connected to bound variables).

- Every agent has exactly "one output" (functionality). In particular, if the agent is a constructor, the principal port is already an output, and all the partitions must be inputs. Conversely, in the case of destructors, we have exactly one output partition among the auxiliary ports, and this partition has to be a singleton.

If $k_1 \dots k_n$ is the arity of a form, every k_i denotes the number of positive ports in the i-th *input* partition. In the case of a destructor, one input partition must be a singleton (since it corresponds to the principal port), so its arity is 0. By convention, in the concrete syntax of IS's, we have supposed that this is always the first partition of the destructor (this assumption is absolutely irrelevant). Summing up, a form with arity $k_1 \dots k_n$ is associated with an operator with $1 + \sum_{i=1}^{n}(k_i + 1)$ ports.

11.1.1.2 Dynamics

Dynamics, in logical systems, is given by the cut elimination process; the idea is that every proof ending into a cut can be simplified into another one by means of some mechanism that is characteristic of that cut (providing, in this way, a rewriting system over proofs). In particular, there exist essentially two kinds of cuts. The most interesting case is the *logical cut*, i.e., a cut between two dual (left-right) introduction rules for the same principal formula. The way such a cut should be eliminated is obviously peculiar to the system, and to the intended semantics of the formula. In all the other cases (structural cuts), Intuitionistic Systems "eliminate" the cut by lifting it in the premise(s) of one of the rules preceding the cut (that becomes the last rule of the new proof). As in the case of λ-calculus, these kind of cuts are essentially "unobservable" in IS's (they are dealt with in the metalevel definition of substitution). So, let us concentrate on logical cuts only.

A typical case of logical cut is that for implication in Intuitionistic Logic.

$$\dfrac{\dfrac{,x : A \vdash t : B}{\vdash \lambda(x.\,t) : A \to B} \qquad \dfrac{\Delta \vdash t' : A \qquad y : B, \Theta \vdash t'' : C}{\Delta, z : A \to B, \Theta \vdash t''[{}^{@(z,t')}/_{y}] : C}}{,\Delta, \Theta \vdash t''[{}^{@(z,t')}/_{y}][{}^{\lambda(x.\,t)}/_{z}] : C}$$

The elimination of the above cut consists in introducing two cuts of

lesser grade (see [Gir89b]). The rewritten proof is:

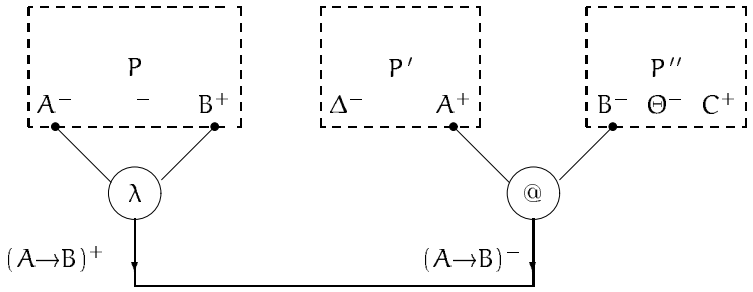
$$\dfrac{\Delta \vdash t' : A \qquad \dfrac{,x : A \vdash t : B \qquad y : B, \Theta \vdash t'' : C}{,x : A, \Theta \vdash t''[^{t}/_{y}] : C}}{,\Delta, \Theta \vdash t''[^{t}/_{y}][^{t'}/_{x}] : C}$$

Since by assumption

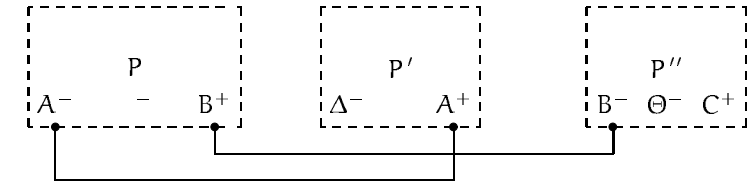
$$t''[^{@(z,t')}/_{y}][^{\lambda(x.\,t)}/_{z}] = t''[^{t}/_{y}][^{t'}/_{x}]$$

this meta-operation on proofs obviously induces the beta-reduction rule in the underlying IS.

Graphically, the above cut elimination rule can be represented as follows:



reduces to:



In general, let L and R be the left and right sequent in the cut-rule, respectively. During the process of cut-elimination, the proofs ending into the premises of L and R must be considered as "black boxes" (only their "interfaces", that is their final sequents are known). During cut-elimination, one can build new proofs out of these black boxes. The unique constraint is the *prohibition of having new hypotheses* in the final sequent of rewritten proof. This has two implications:

(i) the variables bound by L or R (i.e., the auxiliary formulae of the

rules) must be suitably filled in (typically with cuts or introducing *new* rules binding them);

(ii) if a new axiom is introduced by the rewriting, then the hypothesis in the lhs must be consumed (with a cut or by binding it via a logical rule) inside the rewritten proof.

According to statics, a cut in an intuitionistic proof system corresponds to a term of the kind

$$\mathbf{d}(\mathbf{c}(\vec{x}^{\,1}.X_1,\cdots,\vec{x}^{\,m}.X_m),\cdots,\vec{x}^{\,n}.X_n)$$

that is just an IS's redex. The X_i represent the proofs ending into the upper sequents of L and R (the "black boxes" above), and the above conditions on the rewritten proof are obviously reflected in IS's by left linearity and the assumption that right hand sides of rules must be closed expressions.

Example 11.1.7 (Naturals) If we do not like to introduce all naturals as explicit constructors, we could just work with two constructors $\mathtt{0}$ and $\mathtt{succ}$, respectively associated with the following right introduction rules:

$$(\mathtt{nat}, \mathit{right}_0) \quad \vdash \mathtt{0} : \mathtt{nat} \qquad\qquad (\mathtt{nat}, \mathit{right}_S) \quad \frac{\Delta, \vdash n : \mathtt{nat}}{\Delta \vdash \mathtt{succ}(n) : \mathtt{nat}}$$

A typical destructor is $\mathtt{add}$.

$$(\mathtt{nat}, \mathit{left}_{add}) \quad \frac{\Delta \vdash p : \mathtt{nat} \qquad\qquad , y : \mathtt{nat} \vdash t : A}{\Delta, \ , x : \mathtt{nat} \vdash t[^{\mathtt{add}(x,p)}/_y] : A}$$

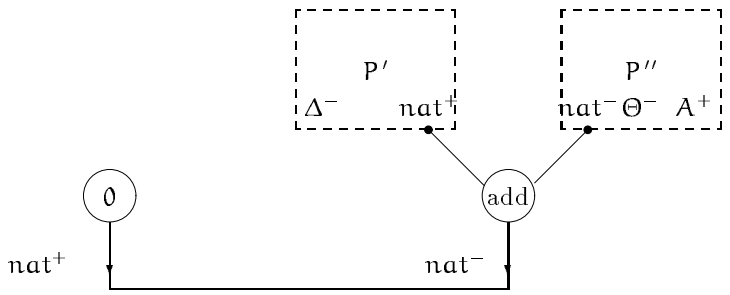
where A can be any type. The following is an example of cut:

$$\dfrac{\vdash \mathtt{0} : \mathtt{nat} \qquad \dfrac{\Delta \vdash p : \mathtt{nat} \qquad y : \mathtt{nat}, \Theta \vdash t : A}{\Delta, x : \mathtt{nat}, \Theta \vdash t[^{\mathtt{add}(x,p)}/_y] : A}}{\Delta, \Theta \vdash t[^{\mathtt{add}(x,p)}/_y][^{\mathtt{0}}/_x] : A}$$

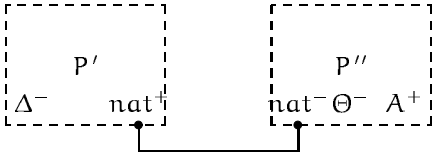
that is simplified into:

$$\frac{\Delta \vdash p : \mathtt{nat} \qquad\qquad y : \mathtt{nat}, \Theta \vdash t : A}{\Delta, \Theta \vdash t[^{p}/_y] : A}$$

The above elimination induces the IS-rule $\mathtt{add}(\mathtt{0}, X) \rightarrow X$, corresponding to the equation $t[^{\mathtt{add}(x,p)}/_y][^{\mathtt{0}}/_x] = t[^{p}/_y]$. Graphically:



reduces to:



Example 11.1.8 (Lists) Lists are defined by means of two constructors `cons` and `nil` of arity 00 and ε, respectively. The typical destructors are `hd` and `tl` of arity 0. In the case of lists of integers, we may write the following introduction rules for the type *natlist*:

$$(natlist, right_{nil}) \quad \vdash \mathtt{nil} : \mathrm{natlist}$$

$$(natlist, right_{cons}) \quad \frac{\Delta \vdash n : \mathrm{nat} \qquad \vdash l : \mathrm{natlist}}{\Delta, \quad \vdash \mathtt{cons}(n, l) : \mathrm{natlist}}$$

$$(natlist, left_{hd}) \quad \frac{, y : \mathrm{nat} \vdash t : A}{, x : \mathrm{natlist} \vdash t[^{\mathtt{hd}(x)}/_{y}] : A}$$

$$(natlist, left_{tl}) \quad \frac{, y : \mathrm{natlist} \vdash t : A}{, x : \mathrm{natlist} \vdash t[^{\mathtt{tl}(x)}/_{y}] : A}$$

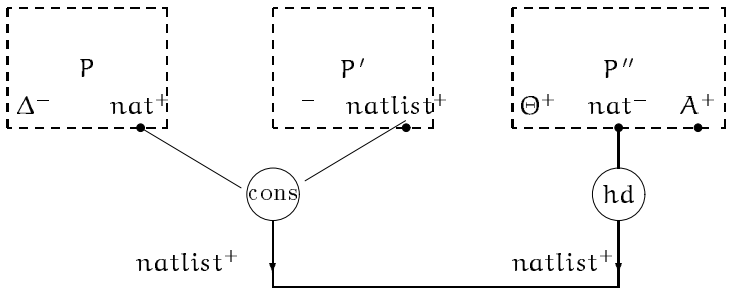
A typical cut is:

$$\frac{\dfrac{\Delta \vdash n : \mathrm{nat} \qquad \vdash l : \mathrm{natlist}}{\Delta, \quad \vdash \mathtt{cons}(n, l) : \mathrm{natlist}} \qquad \dfrac{\Theta, y : \mathrm{nat} \vdash t : A}{\Theta, x : \mathrm{natlist} \vdash t[^{\mathtt{hd}(x)}/_{y}] : A}}{\Delta, \ , \Theta \vdash t[^{\mathtt{hd}(x)}/_{y}][^{\mathtt{cons}(n,l)}/_{x}]}$$

In this case, the cut would be eliminated in the following way

$$\frac{\Delta \vdash n : \mathrm{nat} \qquad \Theta, y : \mathrm{nat} \vdash t : A}{\Delta, \Theta \vdash t[^{n}/_{y}] : A}$$

corresponding to the reduction rule $\mathtt{hd}(\mathtt{cons}(X, Y)) \to X$. Graphically:



reduces to:

Note here, by the way, the phenomenon of garbage creation (P').

All the rewriting rules considered so far are *linear* in their metavariables. An interesting example of non linear rule is provided by the recursion operator μ, defined by:

$$\mu x.M \to M[^{\mu x.M}/_{x}]$$

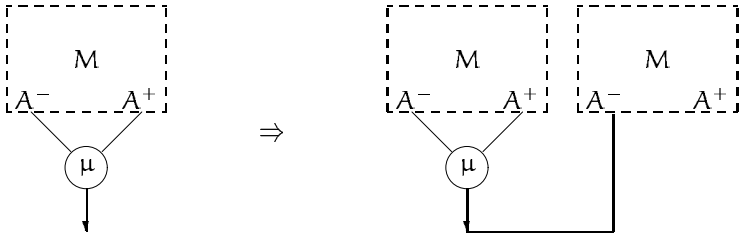
As we discussed in Example 11.1.6, this is a degenerate case of IS-rule, and thus a degenerate case of "cut". The idea is that, given a proof of the kind

$$\frac{\Gamma, x : A \vdash M : A}{\Gamma \vdash \mu x.M : A}$$

it can be rewritten as

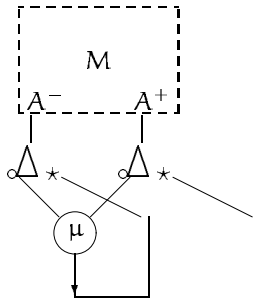
$$\dfrac{\dfrac{\Gamma, x : A \vdash M : A}{\Gamma \vdash \mu x.M : A} \qquad \Gamma, x : A \vdash M : A}{\Gamma, \Gamma \vdash M[^{\mu x.M}/_{x}] : A}$$
$$\Gamma \vdash M[^{\mu x.M}/_{x}] : A$$

Note the double use of the metavariable M in the rule, which is reflected by the double use of the subproof of $\Gamma, x : A \vdash M : A$ in the latter proof. Graphically we have something of the following kind:



Note that the portion of graph corresponding to M has been duplicated. Moreover, all the free variables in M (those marked with in the picture above) must be suitably shared. This means that the subterm M has to be considered as a global entity (a *box*, in Linear Logic terminology), and we loose the possibility to express the rewriting rule as a local operation interacting with the rest of the world only through an interface.

However the locality of rewriting rules can still be recovered by *implementing* IS's in sharing graphs: we have just to use Lamping's fan operators, replacing the graph in the right hand side by a structure of the following kind:



The main problem, that we shall address in the following sections, is to put croissants and brackets in such a way as to guarantee the correct matching between the fan-in and the fan-out, and to avoid conflicts with the internal operator of the box.

11.2 Sharing Graphs Implementation

In this section we shall describe the optimal implementation of Interaction Systems in Sharing Graphs. We shall not give neither the correctness nor the optimality proof, which are quite technical and not very informative. Actually, both these aspects essentially follows by the correctness and optimality of the Sharing Graphs implementation of (the

box of) Linear Logic. Let us also remark that the optimality proof requires the preliminary generalization of the notion (and the theory) of family reduction to IS's. The simplest way to extend this notion is by defining a suitable labeled version of the reduction system. Again, the intuition is simple, but the details are quite involved (see [AL94b]) and we shall skip them†.

The optimal implementation is described as a graph rewriting system, where the nodes of the graph are either control operators (squares, croissants and fans) or syntactical forms of the IS. We shall start with discussing the general encoding of IS-expressions; subsection 11.2.2 will deal with the translation of IS-rewriting rules.

11.2.1 The encoding of IS-expressions

The general idea of the encoding into Sharing Graphs is very simple: each metavariable (that can possibly have a not linear use) is put inside a box (represented in the usual way). For the sake of simplicity, we shall merely consider the paradigmatic case when constructors and destructors have respectively arity $\mathsf{1}$ and $\mathsf{01}$. The other cases are easily derived. The encoding is defined by the translation of Figure 11.2 (as usual we have $[\mathsf{M}] = [\mathsf{M}]_{\mathsf{0}}$).

In this figure, the dangling edges in the bottom represent generic free variables which are not bound by the forms $\mathbf{c}$ and $\mathbf{d}$. As usual, if some bound variable does not occur in the body the corresponding port of the binder is connected to a garbage node.

Note that when we put an argument of a constructor or a destructor inside a box the control operators to be added at the level of the bound variable are different form the control operators to be added to free variables. The reader should intuititively imagine to have a *pseudo-binder* (a λ-abstraction) between the form and the body of the argument; in particular, first, we perfom the abstraction, then, we build the box. Since the pseudo-binder is a ghost, it magically disappears, and we obtain the translation of Figure 11.2. The reason for proceeding in this way will

† The general idea is the following. When a redex is fired, a label α is captured between the destructor and the constructor; this is the label associated with the redex. Then, the rhs of the rewriting rule must be suitably "marked" with α, in order to keep a trace of the history of the creation. Moreover, since in the rhs we may introduce new forms, we must guarantee a property similar to the initial labeling, where all labels are different. This means that all links in the rhs must be marked with a different function of α (we can use sequences of integers, for this purpose).

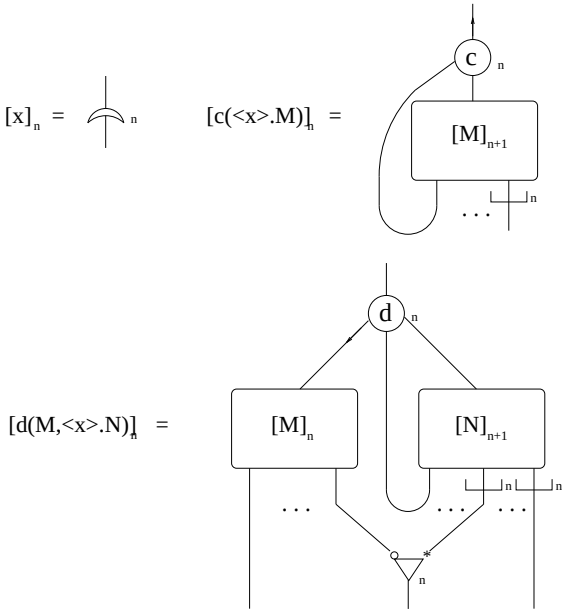


Fig. 11.2. The encoding of IS-expressions

become clear when we will describe the translation of the IS-rewriting rules. At that stage, the ghost-binder will become apparent and will play an essential role during partial evaluation.

11.2.2 The translation of rewriting rules

Rewriting rules may be classified in three groups: control rules, interfacing rules and proper rules. We shall discuss each group separately.

11.2.2.1 Control Rules

These are the well known 12 rules for annihilation and mutual crossing of control operators. These rules provide the general framework for the optimal implementation of the structural part of any IS's.

11.2.2.2 Interfacing Rules

These are the rules which describe the interaction between control operators and forms of the syntax (that is, they describe the interface between the structural and the logical part of IS's). These rules have a

polimorphic nature. We define them by means of schemas, where **f** can be an arbitrary form of the syntax. The rules are drawn in Figure 11.3.

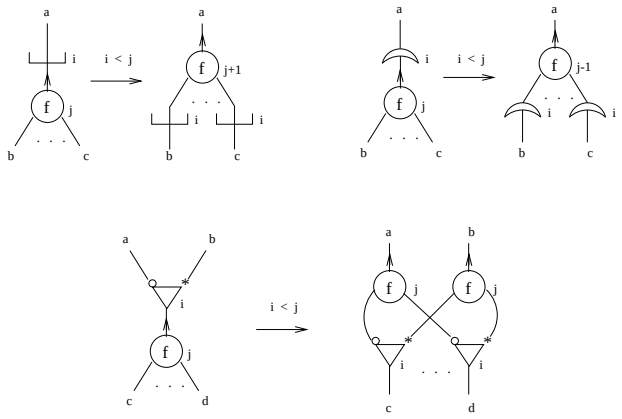


Fig. 11.3. The interfacing rules between control operators and forms.

As you see, interfacing the structural and the logical part of IS at the implementation level is *very* simple. This is a main consequence of the logical nature of IS's, and one of the main motivations for their definition.

11.2.2.3 Proper Rules

These rules describe the interactions between destructors and constructors of the IS's. These are the only rules which are dependent from the particular Interaction System under investigation, and the only ones which deserve some care, in the translation. We shall define the implementation of the rewriting rules in four steps: β-expansion, linearization, translation and partial evaluation.

The idea behind β-expansion and linearization is that of expliciting the "interface" between the new forms which have been possibly introduced in the rhs of the rule, and the metavariables in its lhs. Then, we may essentially translate the rhs as a normal term, just regarding each metavariable as a "black box". Finally, we must partially evaluate the graph obtained in this way, since during β-expansion and linearization we have introduced some "pseudo-operators" which should disappear.

The linearization step is particularly important (β-expansion is just aimed to linearization). It purpose is to obtain an equivalent form of the

rewriting rule where each metavariable is used linearly in the rhs (recall the problem of μ in Example 11.1.6).

(β-**expansion**) The first step is to β-expand all substitutions in the rhs. The aim of this step is to provide a clean vision of all the metavariables in the rhs. For this purpose we shall use two classes of *pseudo-forms*: abstraction $\mathbf{Abs}_n$ and application $\mathbf{App}_n$, for $n \geq 0$. Pseudo-forms are similar to all other forms of the syntax. $\mathbf{Abs}_n$ is a constructor of arity n whilst $\mathbf{App}_n$ is a destructor of arity 0^{n+1}. As the reader could probably imagine, they generalize λ-calculus abstraction and application. Their interaction is expressed by the rule:

$$\mathbf{App}_n(\mathbf{Abs}_n(\langle x_1, \cdots, x_n \rangle . X), Y_1, \cdots, Y_n) \rightarrow X[^{Y_1}/_{x_1}, \cdots, ^{Y_n}/_{x_n}]$$

The step of β-expansion consists in rewriting the rhs of the IS-rule by β-expanding substitutions into interactions of the pseudo-operators $\mathbf{Abs}_n$ and $\mathbf{App}_n$. In particular, after the β-expansion, all metavariables are closed by pseudo binders, i.e. they become expressions of the following kind: $\mathbf{Abs}_n(\vec{x}.M)$.

Example 11.2.1 Consider the rewriting rule for μ:

$$\mu x.M \rightarrow M[^{\mu y.M[^y/_x]}/_x]$$

The β-expansion of the rhs gives the following term:

$$\mathbf{App}(\mathbf{Abs}(x.M), \mu(y.\mathbf{App}(\mathbf{Abs}(x.M), y))))$$

(**linearization**) The next step consists in *linearizing* the rhs w.r.t. the occurrences of expressions $\mathbf{Abs}_n(\vec{x}.X)$. This is obtained by taking, for every metavariable X_i occurring in the left hand side of the IS-rewriting rule (let them be k), a fresh *pseudo-variable* w_i and replacing every occurrence of $\mathbf{Abs}_n(\vec{x}^{\,i}.X_i)$ with w_i. In this way we yield a metaexpression T. Next T is closed w.r.t. the metavariables w_i's, and the (closed) metavariables $\mathbf{Abs}_n(\vec{x}^{\,i}.X_i)$ are passed as arguments to this term. In other words, by linearization, we get a metaexpression of the following kind, where each metavariable occur exactly once (and no substitution is applied to them):

$$\mathbf{App}_k(\mathbf{Abs}_k(\langle w_1, \cdots, w_k \rangle . T), \mathbf{Abs}_{n_1}(\vec{x}^{\,1}.X_1), \cdots, \mathbf{Abs}_{n_k}(\vec{x}^{\,k}.X_k))$$

(n_i is the arity of the metavariable X_i.)

Example 11.2.2 After the linearization step, the rhs of the recursion

rule becomes:

$$\mathbf{App}(\mathbf{Abs}(w.\,\mathbf{App}(w,\,\mu y.\,\mathbf{App}(w,\,y))),\,\mathbf{Abs}(x.\,M))$$

Let us remark that every metavariable in the lhs of the IS-rewriting rule occurs exactly once in the linearized metaexpression yielded by the above procedure, even if it does not occur in the rhs of the IS-rewriting rule. For instance, in the case of conditionals, the linearization of the rhs of $\sharp(\mathbf{T},\,X,\,Y) \to X$ gives

$$\mathbf{App}_2(\mathbf{Abs}_2(\langle w_1,\,w_2\rangle.\,w_1),\,\mathbf{Abs}_0(X),\,\mathbf{Abs}_0(Y)).$$

The actual erasing will be performed in the following steps (see translation and partial evaluation).

(**translation**) This step provides the graphical representation of the rhs of the rule. It is essential that, during the translation, we may consider each subexpression $\mathbf{Abs}_n(\vec{x}.\,X)$ as a "black-box". According to the linearization step, the expression that results will have the shape

$$\mathbf{App}_k(M,\,\mathbf{Abs}_{n_1}(\vec{x}^{\,1}.\,X_{i_1}),\cdots,\mathbf{Abs}_{n_k}(\vec{x}^{\,k}.\,X_{i_k}))$$

The translation of this expression is drawn in Figure 11.4, where, for simplicity, we have assumed $n_i = 1$, for every i. Note that metavariables are not put inside boxes: they *already are* boxes, due to the translation of expressions in Figure 11.2. In particular, no operation around the (unaccessible!) free variables of the instance of the metavariable must be performed.

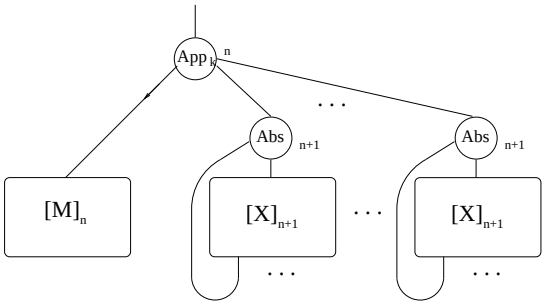


Fig. 11.4. The translation step

Now we can provide some more intuition about our translation in Figure 11.2, and the role of the "ghost-binders". In particular, ghost-binders become apparent in the translation in Figure 11.4: they are

the **Abs** pseudo-forms. The reason for introducing ghost-binder when translating rules, instead of when translating terms, is that we may now partially evaluate the rhs, eliminating all pseudo-forms which have been just introduced for convenience. This is the purpose of the next, final phase. Obviously, all the other pseudo-applications and pseudo-abstractions are translated as usual applications and λ-nodes.

Example 11.2.3 According to the previous rules, the right hand side of the recursion rule has the shape of Figure 11.5 (we implicitly performed some safe optimizations, see Chapter 9)

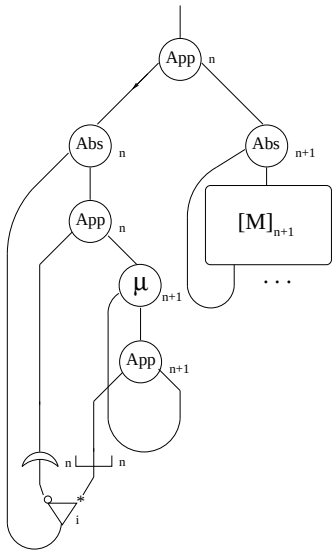
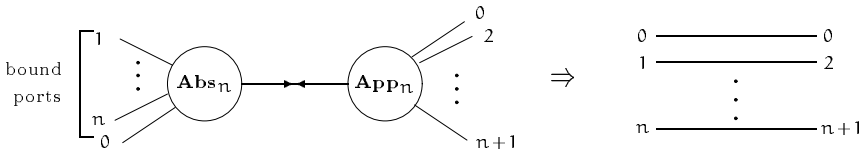


Fig. 11.5. The rhs for the recursion rule.

A final observation before discussing partial evaluation. As already said, some metavariables in the lhs of the IS-rewriting rule could not occur in the rhs (e.g., the case of conditionals). According to the translation of **Abs**, the corresponding pseudo-variable introduced in the linearization step is implemented by a garbage node (since it does not occur in the body of the leftmost outermost **Abs**). This implies that, during the next phase, the corresponding expression will be eventually erased. (**partial evaluation**) The final step is to partially evaluate the term we have obtained after the translation w.r.t. all pseudo operators. The reduction rule for pseudo application and abstraction is



Note that n can be 0. In this case, the rewriting rule simply consists in connecting the two edges coming into the auxiliary ports of $\mathbf{Abs}_0$ and $\mathbf{App}_0$.

It is easy to prove that the partial evaluation of the expressions yielded by the previous translation steps strongly normalizes to a graph without pseudo-forms (essentially, it is a direct consequence of the fact that all pseudo-operators have been created by β-expansions).

Example 11.2.4 By applying the previous technique to our working example of the recursion operator, we finally obtain the optimal reduction rule of Figure 11.6, in Sharing Graphs.

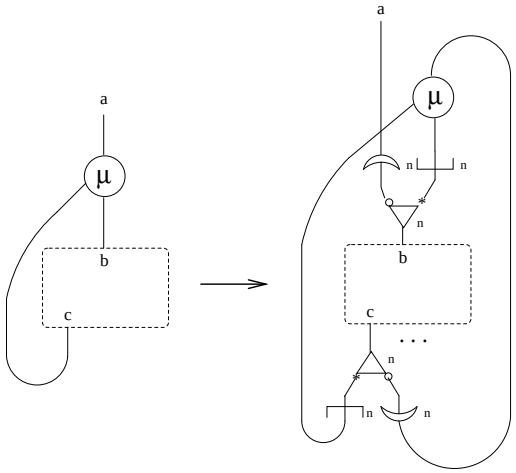


Fig. 11.6. The optimal implementation of recursion in Sharing Graphs.

Remark 11.2.5 The previous translation can be improved by considering special cases of the particular IS's under investigation. In particular, some operators of the syntax could make only a linear use of some of their arguments. For instance, this is the case of the λ-calculus, where the body of the abstraction is treated linearly in β-reduction. These linear arguments may deserve a simpler translation, since there is no need

to put them inside a "box". However, in order to conclude that some operator **f** behaves linearly over one of its arguments we must examine *all* the interaction rules involving **f**. Thus, the choice of the rewriting system and its forms may have a big impact on the practical efficiency of the implementation.

12
The Bologna Optimal Higher-order Machine

This chapter describes the main architecture of the Bologna Optimal Higher-order Machine (BOHM).

BOHM is a prototype implementation of an extension of Lamping's optimal graph reduction technique. Its source language is a sugared λ-calculus enriched with booleans, integers, lists and basic operations on these data types.

The operational semantics of BOHM will be given in the form of an Interaction System. In particular, this means that:

(i) All syntactical operators will be represented as nodes in a graph.
(ii) Nodes will be divided into *constructors* and *destructors*.
(iii) Reduction will be expressed as a local interaction (graph rewriting) between constructor-destructor pairs.

The reader should not be particularly surprised of this, for we already described in detail the tight relation between Interaction Systems and optimal implementations (see Chapter 11).

From the reduction point of view, BOHM is a *lazy* system. Namely, given an input term, it reduces the corresponding graph until the top-most node (i.e., the one connected to the "root") becomes a constructor.

BOHM is just a high level interpreter written in C. Its purpose was not to be a real implementation, but to provide some empirical evidence about the feasibility of Lamping's technique. Because of this, particular care has been devoted in supporting the user with a large set of data on the resources used by computations (user and system time, number of interactions, storage allocation, garbage collection, and so on).

The source code is available by anonymous ftp at `ftp.cs.unibo.it`, in the directory `/pub/asperti`, under the name `BOHM.tar.Z` (compressed tar format).

12.1 Source Language

As described in Chapter 11, Lamping's optimal implementation technique can be generalized to Interaction Systems. This means that:

(i) It is possible to translate an arbitrary IS-expression into a suitable graph representation using Lamping's control operators for representing structural information (sharing), and new graphical forms for the logical part of the system (in particular, we shall have a new graphical form for each operator in the signature).

(ii) Any IS-rewriting rule has an associated graph rewriting rule, corresponding to the local interaction of two forms at their principal ports.

(iii) The graph rewriting rules are correct and optimal w.r.t. the given Interaction System.

The main issue of this extension is that Interaction Systems are powerful enough to encompass all typical constructs of a "core" functional language.

The source language of BOHM is a sugared λ-calculus enriched with some primitive data types (booleans, integers, lists) equipped by the usual operations, and two more control flow constructs: an explicit fixpoint operator (`rec`) and a conditional `if-then-else` statement.

The syntax of the language accepted by BOHM is given in Figure 12.1.

12.2 Reduction

The operational semantics of the language is given in the form of an Interaction System. Therefore, all the rewriting rules will thus be written as destructor-constructor interactions. In some cases, this will however require the introduction of a few auxiliary destructors.

The full list of constructors, destructors, and their respective arities, is given below:

constructors $\lambda : 1$; $\mathsf{true} : \varepsilon$; $\mathsf{false} : \varepsilon$; $\mathsf{m} : \varepsilon$, for each integer m; $\mathsf{nil} : \varepsilon$; $\mathsf{cons} : 00$.

destructors $@ : 00$; $+ : 00$; $+_{\mathsf{m}} : 0$; $- : 00$; $-_{\mathsf{m}} : 0$; $* : 00$; $*_{\mathsf{m}} : 0$; $\mathsf{div} : 00$; $\mathsf{div}_{\mathsf{m}} : 0$; $\mathsf{mod} : 00$; $\mathsf{mod}_{\mathsf{m}} : 0$; $\mathsf{and} : 00$; $\mathsf{or} : 00$; $\mathsf{not} : 0$; $== : 00$; $==_{\mathsf{m}} : 0$; $> : 00$; $>_{\mathsf{m}} : 0$; $< : 00$; $<_{\mathsf{m}} : 0$; $>= : 00$; $>=_{\mathsf{m}} : 0$; $<= : 00$; $<=_{\mathsf{m}} : 0$; $<> : 00$; $<>_{\mathsf{m}} : 0$; $\mathsf{head} : 0$; $\mathsf{tail} : 0$; $\mathsf{if\_then\_else} : 000$.

```
⟨expr⟩     ::= true
             | false
             | ⟨num_const⟩
             | ⟨identifier⟩
             | (⟨applist⟩)
             | \ ⟨identifier⟩ . ⟨expr⟩
             | let ⟨identifier⟩ = ⟨expr⟩ in ⟨expr⟩
             | rec ⟨identifier⟩ = ⟨expr⟩
             | if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩
             | ⟨expr⟩ and ⟨expr⟩
             | ⟨expr⟩ or ⟨expr⟩
             | not ⟨expr⟩
             | ⟨expr⟩⟨relop⟩⟨expr⟩
             | ⟨expr⟩⟨mathop⟩⟨expr⟩
             | ⟨list⟩
             | cons (⟨expr⟩, ⟨expr⟩)
             | head (⟨expr⟩)
             | tail (⟨expr⟩)
             | isnil (⟨expr⟩)
⟨list⟩     ::= nil
             | [⟨exprlist⟩]
⟨exprlist⟩ ::= ⟨expr⟩
             | ⟨expr⟩, ⟨exprlist⟩
⟨applist⟩  ::= ⟨expr⟩
             | ⟨applist⟩⟨expr⟩
⟨relop⟩    ::= < | == | > | <= | >= | <>
⟨mathop⟩   ::= + | - | * | div | mod
```

Fig. 12.1. BOHM's syntax

As already remarked, rec is the unique exception to the previous classification. Indeed, rec is by itself a constructor-destructor pair.

Here are the proper interactions of the rewriting system:

- *Beta-reduction*
  $(\lambda x.\, M\ N) \rightarrow M[N/x]$

- *Recursive Definition*
  $\mathsf{rec}\ x = M \rightarrow M[(\mathsf{rec}\ x = M)/x]$

- *Arithmetical Operators*

  $m + M \rightarrow +_m(M)$
  $+_m(n) \rightarrow p$ where p is the sum of m and n
  $m - M \rightarrow -_m(M)$
  $-_m(n) \rightarrow p$ where p is the difference of m and n
  $m * M \rightarrow *_m(M)$
  $*_m(n) \rightarrow p$ where p is the product of m and n
  $m \text{ div } M \rightarrow \text{div}_m(M)$
  $\text{div}_m(n) \rightarrow p$ where p is the quotient of the division of m by n
  $m \text{ mod } M \rightarrow \text{mod}_m(M)$
  $\text{mod}_m(n) \rightarrow p$ where p is the rest of the division of m by n

- *Boolean Operators*

  true and $M \rightarrow M$
  false and $M \rightarrow$ false
  true or $M \rightarrow$ true
  false or $M \rightarrow M$
  not false $\rightarrow$ true
  not true $\rightarrow$ false

- *Relational Operators*

  $m == M \rightarrow ==_m(M)$
  $==_m(m) \rightarrow$ true
  $]isf==_m(n) \rightarrow$ false if $n \neq m$
  $m < M \rightarrow <_m(M)$
  $<_m(n) \rightarrow$ true if m is less than n
  $<_m(n) \rightarrow$ false if m is not less than n
  $m > M \rightarrow >_m(M)$
  $>_m(n) \rightarrow$ true if m is greater than n
  $>_m(n) \rightarrow$ false if m is not greater n
  $m <= M \rightarrow <=_m(M)$
  $<=_m(n) \rightarrow$ true if m is not greater than n
  $<=_m(n) \rightarrow$ false if m is greater than n
  $m >= M \rightarrow >=_m(M)$
  $>=_m(n) \rightarrow$ true if m is not less than n
  $>=_m(n) \rightarrow$ false if m is less than n
  $m <> M \rightarrow <>_m(M)$
  $<>_m(m) \rightarrow$ false
  $<>_m(n) \rightarrow$ true if $n \neq m$

- *Operator for list manipulation*
  $\mathsf{head}(\mathsf{cons}(\mathsf{M},\mathsf{N})) \rightarrow \mathsf{M}$
  $\mathsf{tail}(\mathsf{cons}(\mathsf{M},\mathsf{N})) \rightarrow \mathsf{N}$
  $\mathsf{tail}(\mathsf{nil}) \rightarrow \mathsf{nil}$
  $\mathsf{isnil}(\mathsf{cons}(\mathsf{M},\mathsf{N})) \rightarrow \mathsf{false}$
  $\mathsf{isnil}(\mathsf{nil}) \rightarrow \mathsf{true}$

- *The control flow operator if-then-else*
  $\mathsf{if\ true\ then\ M\ else\ N} \rightarrow \mathsf{M}$
  $\mathsf{if\ false\ then\ M\ else\ N} \rightarrow \mathsf{N}$

12.2.1 Graph Encoding

BOHM's control operators for sharing are a compromise between Lamping's original ones (square bracket, croissant and fan) and Guerrini's *multiplexers*. In particular, it uses two control operators: the *triangle* and the *fan*, that should be respectively understood as a mux with one or two auxiliary ports (see Figure 12.2).

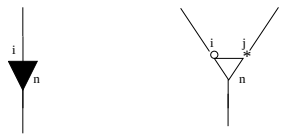


Fig. 12.2. BOHM's *triangle* and *fan*

Similarly to a *mux*, each auxiliary port has an associated weight. So, a croissant is just a triangle with weight -1, a square bracket is a triangle with weight 1 and Lamping's fan is BOHM's fan with weight 0 on both auxiliary ports.

The advantage of triangles and generalized fans, is that they allow to compress a long sequence of control operators into a single node. In particular, *when the lower operator is safe*, we can apply the "merging" rules in Figure 12.3.

Practically, on typical examples, the introduction of these mux-like operators may easily speed up the reduction time of an order of magnitude, with respect to Lamping's algorithm. On the other side, we made several attempts to implement Guerrini's multiplexers, but none of them provided the expected results. The reason is that handling operators with a dynamic number of auxiliary ports introduces a computational overhead

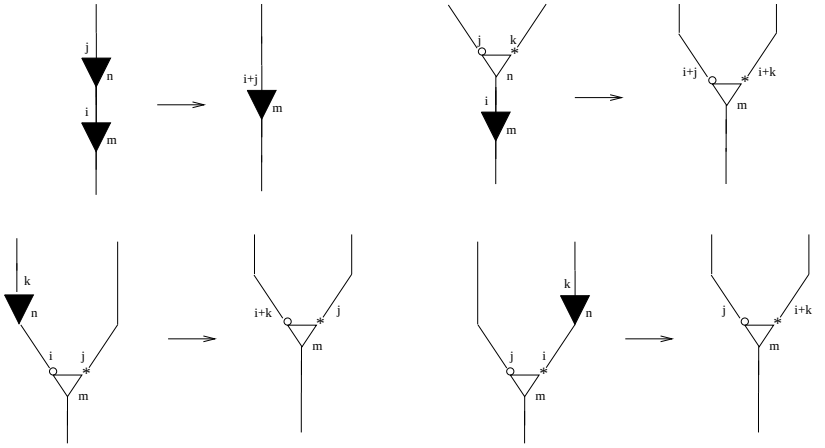


Fig. 12.3. Merging rules — Proviso: $\mathfrak{m} \leq \mathfrak{n} \leq \mathfrak{m}+\mathfrak{i}$, and lower operator *safe*.

that very often is not compensated by the diminution of the number of interactions. Since moreover the code gets much more contrived, we prefered to work with fixed-arity operators.

Having explained the kind of control operators used in BOHM, we may now give the initial translation of input terms into their graphical representation.

As usual, any term N with $\mathfrak{n}$ free variables will be represented by a graph with $\mathfrak{n}+1$ entries (free edges): $\mathfrak{n}$ for the free variables (the inputs), and one for the "root" of the term (the output).

The translation starts at level 0 (see Figure 12.4). For the sake of clarity and for implementation reasons, we add an explicit *root* node at the top of the term. Moreover, we are implicitly assuming that all input terms are closed.

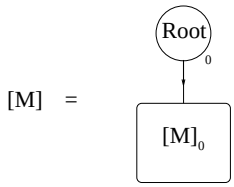


Fig. 12.4. Starting translation rule.

Remark 12.2.1 Actually, there is a global construct `def` for building

up a global environment in which names are associated to expressions (see section 12.2.1.2). According to this, "closed" means that any free variable in an evaluating term must correspond to the name of some previous defined expression.

The general translation rules are described in Figure 12.5 (only some typical examples are depicted). They are a mere specialization of the general translation method for Interaction Systems to our particular source language (and to our choice of control operators). However, the translation has been optimized in several places to take advantage of the *linear* behavior of most part of interaction rules.

The only term that deserves a special comment is the fix-point operator rec $x = M$. According to the usual translation of Interaction System, it should correspond to the graph in Figure 12.6(a). However, we can just get rid of the explicit μ node, adopting the simpler translation in Figure 12.6(b). The latter encoding has been suggested by Juliusz Chroboczek; in this way, reduction is twice faster on highly recursive terms.

We leave as an exercise to the reader to make sense of this translation. Let us just remark that it also modifies the notion of family of redexes (and thus of optimality). Let us consider for instance the following definition of the factorial function:

```
rec fact =
       \n.if n == 0 then 1
                    else (fact (n-1));;
```

With the first encoding, the computation of (fact n) requires n β-reductions. With Chroboczek's encoding, just one (1) β-reduction is performed. The reason is that with the former encoding each recursive application of fact to its argument is considered as a reducible expression that is *created* by firing the fix-point rule (so, they cannot be shared). In the second case, the fix-point operator is "transparent"; there is no explicit rule for μ, so no new redex can be created by its "application".

12.2.1.1 Implementation Issues

The graph created by the previous rules is not oriented (you should not confuse the arrow denoting the principal port of a form with an oriented edge). For this reason, and in order to facilitate graph rewriting, all edges will be represented by a double connection between nodes. In particular, for each port of a syntactical form F we must know the other

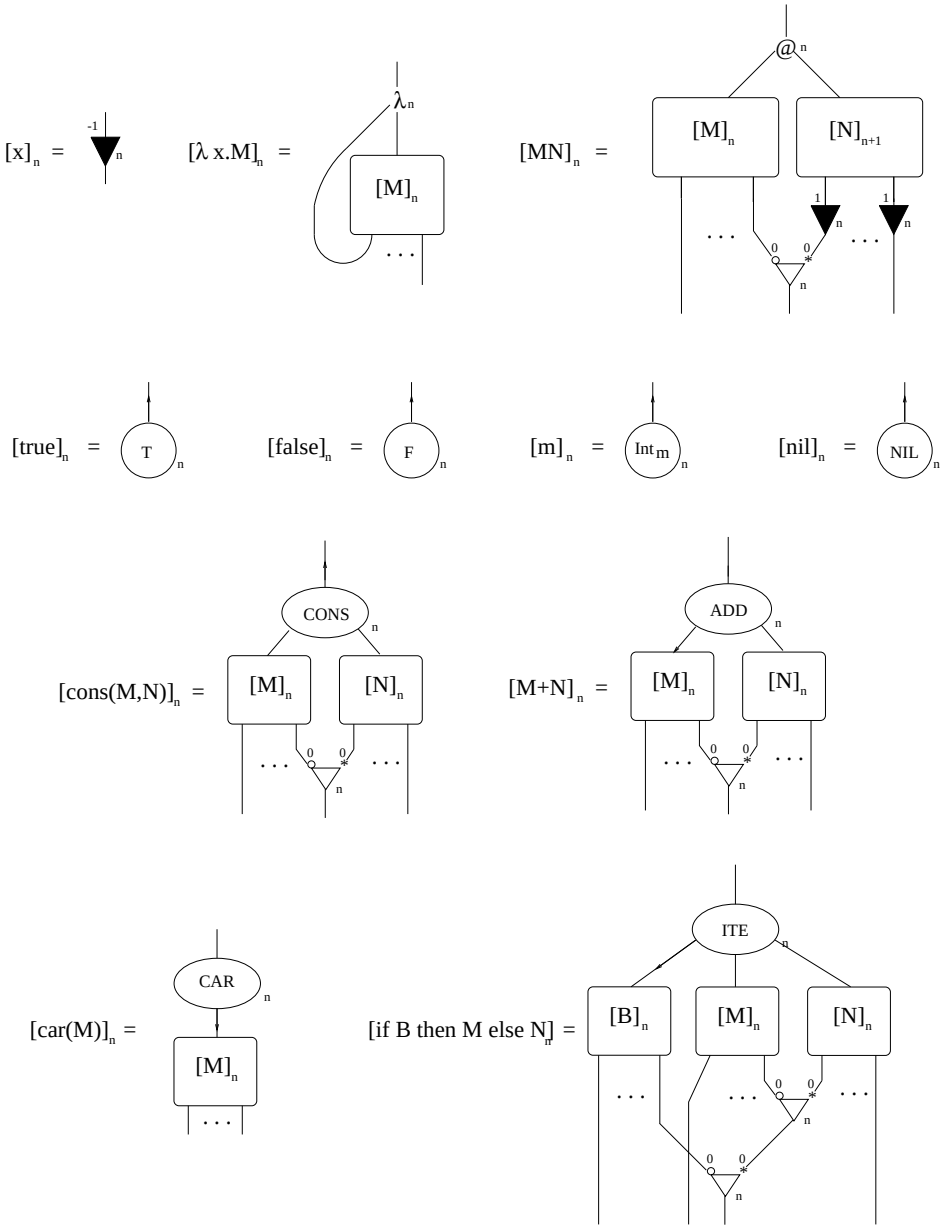


Fig. 12.5. Initial translation.

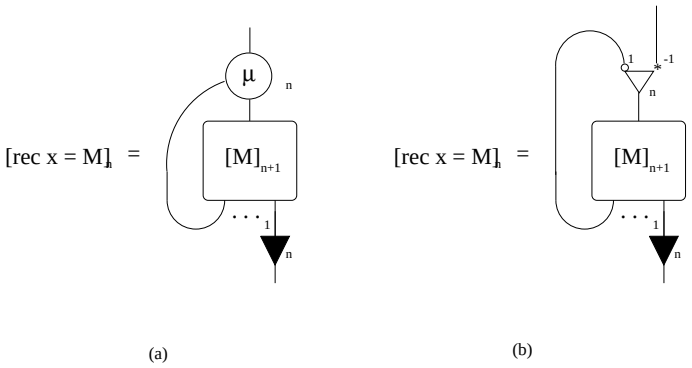


Fig. 12.6. Recursion.

form F′ which is connected to it at that port, and also to which port of F′ it is connected to.

Obviously, each form has also a name (FAN, APP, ADD, ...), and an index. The typical representation of a ternary form may be thus described by the following struct in C.

```
typedef struct form
   {
      int   name,
                  /* name of the form       */
                  /* (FAN, APP, ADD,        */
                  /*  SUB, EQ, ...)         */
            index;
                  /* index of the form */
            int   nport[3];
                  /* numbers of the ports of adjacent */
                  /* forms where the three ports      */
                  /* of this form are connected to    */
            struct form *nform[3];
                  /* pointers to the forms            */
                  /* where the three ports            */
                  /* of this form are connected to    */
   }  FORM;
```

Given a pointer f to a form, the field `f->nform[i]` will denote the next form g connected with f at port i. Similarly, `f->nport[i]` says to

which port of `g` the form `f` is connected. In particular, we always have that `(f->nform[i])->nform[f->nport[i]] == f`.

Remark 12.2.2 By convention, the principal port of a form has always number 0.

Obviously, different sets of nodes require their own specific informations. For instance, some arithmetical nodes need a numerical value; control nodes need a "safeness" tag and "weights" for the auxiliary ports; and so on. Actually, the large *dimension* of FORMS is the main implementation problem of BOHM. At the same time, let us note that all unary constructors (integers, true, false, and nil) do not require an explicit form. In particular, their value can be directly stored in the form to which they are connected, instead of a pointer to them.

Even if the double connection between FORMS introduces some redundancy, it allows to navigate the graph in a very easy way. Indeed, we should avoid a systematic use of bi-links, restricting them to particular edges of the graphs (typically, those representing cut or axioms). Although this solution could reasonably save some space and could also improve performances, it looks more involved to implement; thus, we finally rejected it in designing the prototype.

As a simple example, let us see the function that connects together two forms at two specified ports (this is the most frequently used function of BOHM; each atomic interaction calls `connect` 3-4 times, in average.

```
 /* the following function connects together the port */
 /* portf1 of form1 to the port portf2 of form2 */

connect(form1,portf1,form2,portf2)
       FORM       *form1;
       int        portf1;
       FORM       *form2;
       int        portf2;
{
       form1->nport[portf1] = portf2;
       form1->nform[portf1] = form2;
       form2->nport[portf2] = portf1;
       form2->nform[portf2] = form1;
}
```

12.2.1.2 Global Definitions

The syntactical construct:

$$\texttt{def}\ \langle\text{identifier}\rangle\ =\ \langle\text{expr}\rangle$$

allows the user to declare an expression as a global definition, binding it with an identifier name.

From the point of view of reduction, there are two possible ways for handling global definitions: *sharing* and *copying*.

The first version of BOHM, was based on a sharing approach. In particular, every time we had a new declaration of a term, the graph corresponding to the term was built up and closed with a ROOT form pointed to by the identifier in the symbol table. Then, using a previously defined global expression, we simply attached a FAN at the top of the corresponding graph, implicitly creating a new instance of the expression. In this way, the global definition was shared by all its instances. Note that this also implied a minor change in the translation of terms. In fact, since a global expression is a sharable data, its graph must be created starting at level 1, and not 0 as for usual expressions.

The previous handling of global definitions was however responsible for an odd operational behavior: the second time we evaluated an expression containing global identifiers, the reduction time could be much smaller than the first time. The reason is that the second call took advantage of all partial evaluations that the first call performed on global expressions.

To share global expressions allowed thus to profit of the shared partial evaluation of global terms, exploiting sharing in great extend in accord to the philosophy of optimal reduction. However, in many cases, it also introduced a substantial trade off from the point of view of memory allocation. In fact, the partially evaluated global expression might usually need *much more space* than the original term. For this reason, the new versions of BOHM adopt instead a copy technique: every time we use a global name, a new local instance of the corresponding graph is created.

12.2.2 Graph Reduction

Once the graph representing an expression has been built, the reduction proceeds according to the graph rewriting rules in the next pictures. The rules in Figure 12.7 and Figure 12.8 are the instantiation of the general paradigm for Interaction Systems to our case. Their reading is completely straightforward.

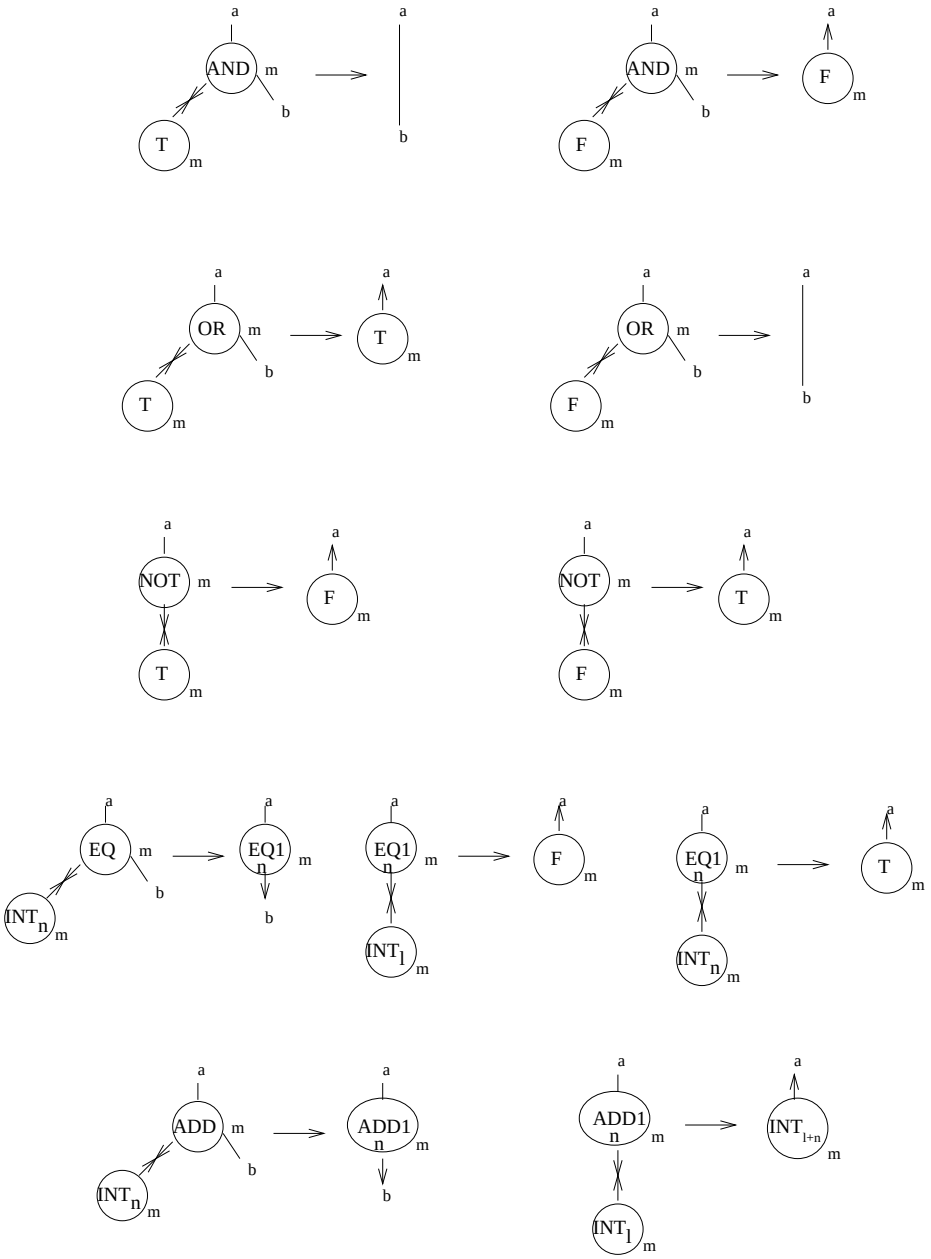


Fig. 12.7. BOHM's interaction rules: logical rules I

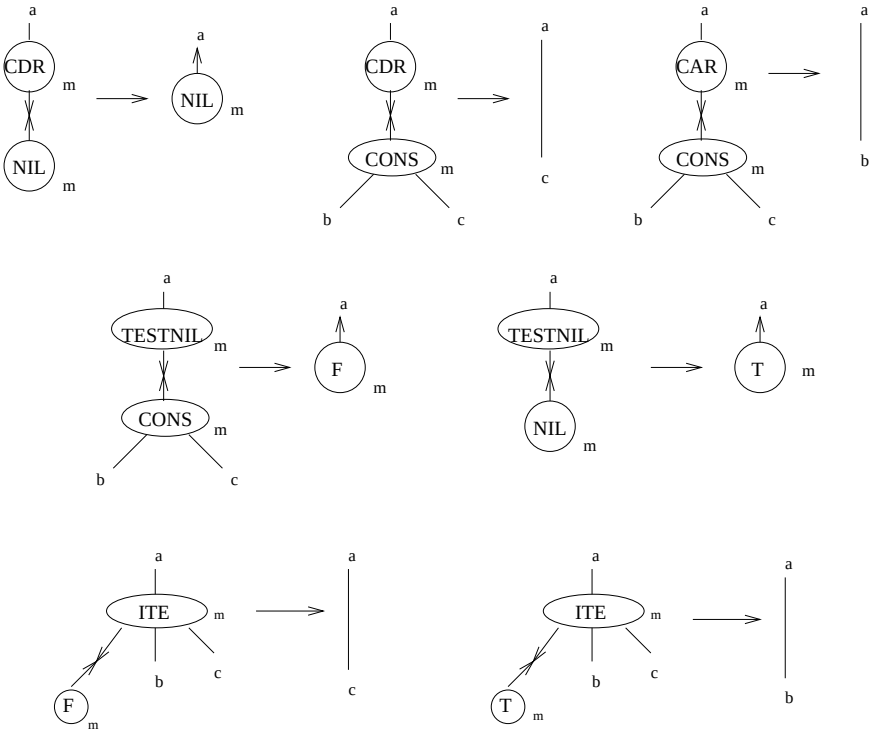
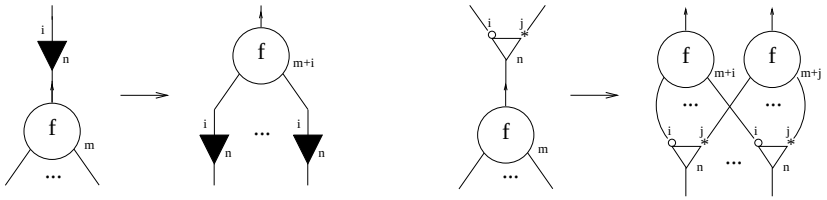


Fig. 12.8. BOHM's interaction rules: logical rules II

The rules in Figure 12.9 cover instead the interactions between control operators (triangle and fans) and proper nodes; they are the obvious generalization of the usual ones. We omit to draw the usual annihilation rules.



n < m

Fig. 12.9. BOHM's interaction rules: triangle and fan

12.2.2.1 Implementation Issues

BOHM is *lazy*. It does not perform the full reduction of a term, it rather stops when the topmost operator in the graph becomes a constructor. This halting situation is easily recognized, since it means that the root of the term becomes *directly* connected to some operator f at its *principal port* (i.e., by convention, the port with number 0). In this case, the name of f is the result of the computation.

Reduction proceeds according to the following idea. We start looking for the leftmost outermost redex. In particular, starting from the root of the term, we traverse any operator that we reach at an auxiliary port *always exiting from its principal port*, until we eventually reach an operator f at its principal port. Now two cases are possible: either the previous operator was the root node (and this is the halting case), or we have found a redex (for the last two operators are connected at their principal ports). In this case, the redex is fired by applying the associated graph reduction rule, and we start the process again.

In order to avoid to rescan the term from the root every time, it is convenient to push all operators leading to the first redex on a stack (similar to the stack of the G-machine). After firing a redex it is enough to pop the last element from this stack, and start searching for the next redex from this node.

The following piece of code is the main loop of the reduction machine:

```
reduce_term(root)
     FORM      *root;
{
     FORM      *f1,
               *f2,
               *erase;
     int       type_error;

     type_error = FALSE;
     f1 = lo_redex(root);
     f2 = f1->nform[0];
     while ((f1 != root) && (!type_error))
        {
           if (f1->index <= f2->index)
               reduce_redex(f1,f2);
           else
               reduce_redex(f2,f1);
```

```
            f1 = lo_redex(pop());
            f2 = f1->nform[0];
         }
      if(!type_error) rdbk(root);
}
```

where `lo_redex` is defined as follows:

```
FORM *lo_redex(f)
     FORM    *f;
{
     FORM    *temp;

     temp = f;
     while (temp->nport[0] != 0)
        {
           push(temp);
           temp = temp->nform[0];
        }
     return temp;
}
```

`reduce_redex(f1,f2)` is just a big switch based on the name of f1 and f2 (and their index). For example, here is the piece of code in the case the two forms have the same index and the first form f1 is AND:

```
case AND:
   switch(f2->name)
      {
          case F:
              connect(f1->nform[1],
                      f1->nport[1],
                      f2,
                      0);
              myfree(f1);
              break;

          case T:
              connect(f1->nform[1],
                      f1->nport[1],
                      f1->nform[2],
                      f1->nport[2]);
```

```
            myfree(f1);
            myfree(f2);
            break;

        default:
            printf("--->   type error\n");
            type_error = TRUE;
            break;
    }
```

12.3 Garbage Collection

In pure λ-calculus, the creation of garbage is related to the presence of abstraction nodes whose bound variable does not appear in the body. As we have already seen, such situations may be represented by introducing a suitable *garbage* node connected to the bound port of the λ-node. For instance, consider the term $(\lambda x.N)M$, where x does not appear in N (see Figure 12.10). After the β-reduction, the term M appears disconnected from the main graph, becoming garbage.

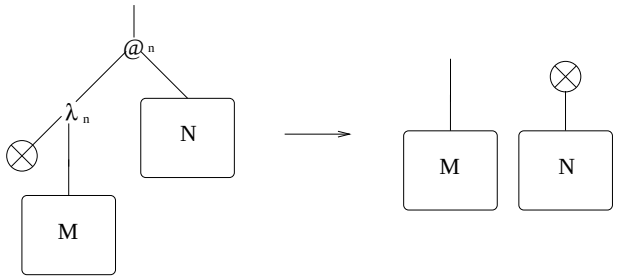


Fig. 12.10. Creation of garbage

Unfortunately, things are usually more complex. In fact, M might share some subterms with the main graph. Therefore, these subterms cannot be regarded as garbage. Moreover, the whole term M would remain connected to the main graph through that shared subterms.

As far as we consider pure lambda calculus, garbage collection is not really impelling; on the contrary, it becomes of dramatical importance in the enriched language of BOHM. As a matter of fact, a lot of BOHM's rewriting rules create garbage. A typical case is the conditional expression if B then M else N. If B is true (false), the subterm N (respectively

M) is garbage. Analogously, the evaluation of the boolean expression false and B immediately reduces to false, and B becomes garbage. Other important cases concern lists; for instance, after an application of the operator CAR, which selects the first element of a list, the tail of the list can be discarded.

In the context of optimal reduction, it looks difficult to implement an efficient garbage collection procedure using traditional *mark and sweep* algorithms. Indeed:

(i) Marking the active elements would require a complex visit of the graph, based on the so called context semantics, and the complexity of such a visit could be exponential in the size of the graph.

(ii) We could limit collection of garbage to physically disconnected subgraph. However, particularly with BOHM's lazy strategy, garbage collection would be delayed very farther in the reduction of the term, causing a possible explosion of garbage.

The second point above should probably deserve some more words. Consider for instance the case of a shared list (a fan over a cons). Suppose moreover that one branch of the fan has become garbage (i.e., we have a garbage operator at one auxiliary port of the fan). This situation occurs very often with BOHM's rules. Obviously, we would like to eliminate the fan and the garbage operator, attaching the list to the only active branch of the fan. However, adopting the policy in the second point above, we could not collect any garbage until we finished duplicating the whole list. On the contrary, in optimal reduction, garbage should always be collected *as soon as possible*.

As pointed out by Lamping, an efficient way to integrate a garbage collector in the optimal graph reduction technique is based on local interactions of erase operators ($\otimes$). The basic idea is that the erase operators are propagated along the graph, collecting every node encountered along their walk.

In BOHM, all erase operators are maintained in an appropriate data structure (a list). When garbage collection is activated each of them interacts with the forms it can erase, adding new erase operators to the list. The process ends when no interaction for any eraser in the list is possible.

The key point is that there are situations in which no erasing interaction is allowed; typically, when an erase operator reaches the binding

port of an abstraction, or an auxiliary port of an unsafe fan. This latter case is particularly important; therefore let us analyze it in full details.

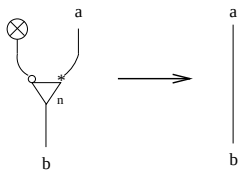


Fig. 12.11. Naive garbage collection of a fan

When a garbage operator reaches a fan at an auxiliary port, one would be tempted to apply the rule in Figure 12.11. Indeed, one of the two branches of the information shared by the fan has become garbage, and the obvious solution would seem to simply drop this branch. However, this rule is not correct in general, due the problem described in Figure 12.12. That is, if we allow a fan-in(-out) to interact with an erase operator according to the rule in Figure 12.11, we could preclude to some "paired" fan-out(-in) the possibility to annihilate with the erased fan.

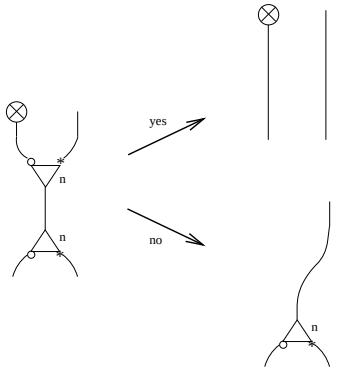


Fig. 12.12. Critical pair created by the rule in Figure 12.11

Luckily, as we already know, the rule is perfectly correct for safe fan (for the trivial reason that no "paired" fan-out could possibly exist, in this case).

Another interesting point is the choice of an appropriate strategy for activating the garbage collection procedure. Our empirical studies seem to suggest that the best strategy is to call the garbage collector every time some reduction rule generates new garbage, i.e., as soon as the

garbage is created. Operating in this way, we obtain two important advantages:

- the size of the graph is always kept to a minimum along the reduction;
- collecting garbage as soon as possible prevents that it becomes involved in useless interactions, resulting in a sensible improvement in performance.

Figure 12.13 contains the whole set of garbage rules for pure lambda calculus. These rules generalize in the obvious way to the other forms of the syntax. In particular, a garbage operator can efface any syntactical form, no matter at which port it reaches the form, unless it is a bound port. In this case it stops there. After collecting the form, we broadcast the garbage operator to all other ports of it, starting the process again.

Remark 12.3.1 The erase node has no index. Moreover, its interactions are independent from the index of the adjacent node.

We stress that the last two rules for fans may be only applied in the case in which the fan is safe, otherwise no interaction is allowed, and the erase operator is stopped. Moreover, let us note, that these rules may create triangles with weight 0. Now, since a triangle of such a kind neither modify the indexes of the nodes it crosses, nor has any sharing effect, it can be always erased from the graph without any consequence on the result of the reduction. However, you must take into account the possibility that the original fan (and thus the triangle) was in the "leftmost-outermost-redex" *stack*. If this is the case, to erase the triangle from the graph we should consistently remove it from the stack too, that is however a complex operation. The simplest solution is to look for triangles with weight zero when we pop elements from the stack, erasing them at that moment. As an example of the subtlety of this problem, we overlooked it in our first implementation and, believe it, the bug was *really difficult* to find.

Although the previous garbage collection procedure works pretty well in practice, it gets in troubles with particularly looping situations. For instance, the whole graph in Figure 12.14 is garbage, but no garbage collection rule may be applied to erase it.

For this reason, BOHM periodically looks for completely disconnected subgraphs (typically, this operation is performed immediately before the reduction of a new input term).

To conclude this introduction to garbage collection, we want to stress

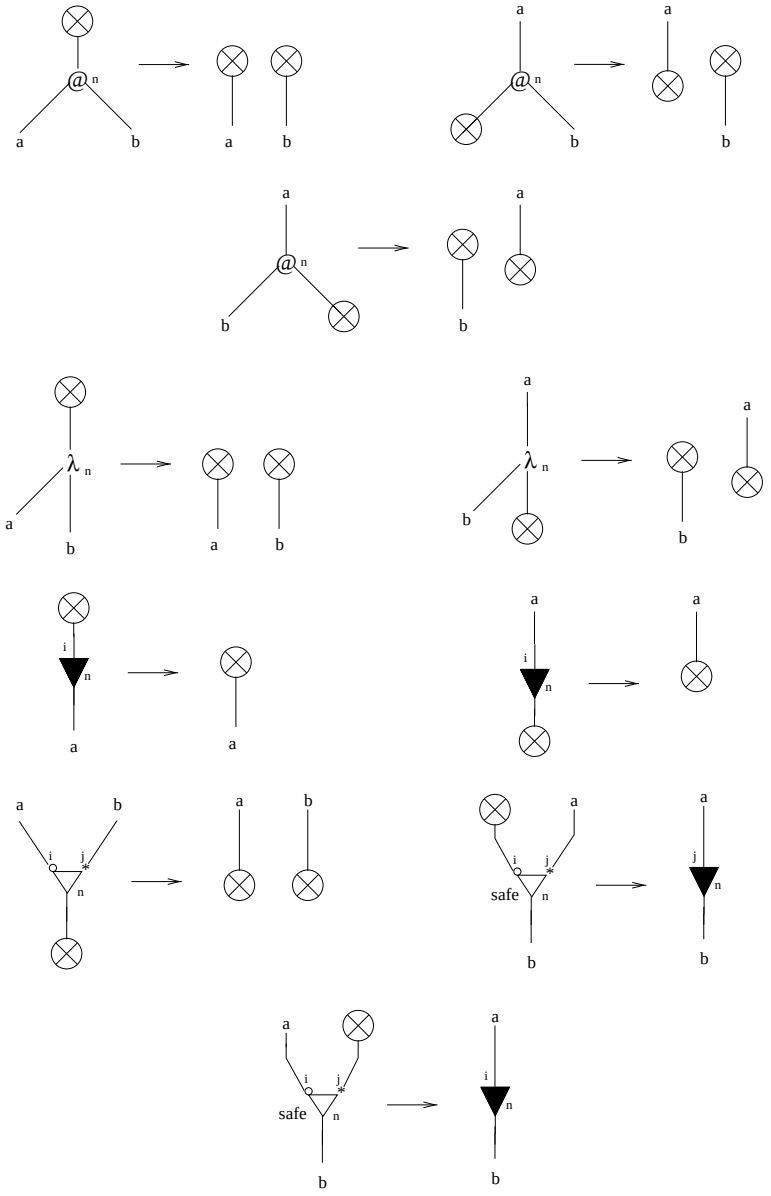


Fig. 12.13. Garbage collection rules

that the theoretical and practical aspects of garbage collection in opti-

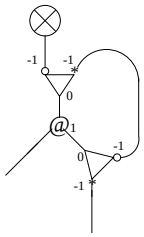


Fig. 12.14. Garbage that cannot be easily collected

mal reduction are still among the most relevant open problems of this discipline, and surely deserves further investigation.

12.3.1 Implementation Issues

To improve the efficiency of the garbage collection procedure, three new nodes have been introduced. They do not represent neither new syntactical elements, nor new control operators; these new forms are just a sort of abbreviation for particular configurations of other forms. In particular (see also Figure 12.15):

- UNS_FAN1 represents a configuration where an erase operator is "blocked" on the "first" auxiliary port of an unsafe fan.
- UNS_FAN2 is similar to the previous one, but relative to the "second" auxiliary port.
- λ-unb represents an abstraction node connected at its bound port with an erase operator.

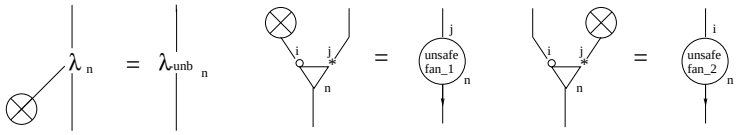


Fig. 12.15. Some new forms incorporating the erase operator

The introduction of these new forms has two important advantages:

(i) All erase nodes which cannot be propagated any further are merged into their adjacent nodes. Therefore, it is easy to recognize when garbage collection is possible: an erase operator is physically present in the graph if and only if it is "active".

(ii) The forms UNS_FAN1 and UNS_FAN2 prevent many useless duplications and garbage collection calls during reduction.

In accord with the idea that BOHM is a sort of testbed. The user can select among three garbage collection strategies and compare the corresponding performances. The possibilities are:

1) Maximum garbage collection. The garbage collector is activated as soon as new garbage is created.
2) Garbage collection depending on memory occupation. In this case, garbage collection is attempted as soon as the total number of allocated nodes reaches a certain bound chosen by the user.
3) No garbage collection.

The user can select one of the previous choice at start time, calling the program with the option *-s*, or at a many moment during the session, invoking a suitable menu via the directive `#option`.

12.3.2 Examples and Benchmarks

In this section, we give a few typical benchmarks about garbage collection.

We start with some examples in pure λ-calculus. In particular, we consider two "primitive recursive" versions of the factorial and fibonacci functions on Church integers. Since the computation stops at weak head normal form, we shall supply some extra-parameters (identities), in order to get an interesting computation. Here is the source code (these definitions can be also found in the file `examples/purelambda` in BOHM's main directory):

```
def I = \x.x;;
def zero = \x.\y.y;;
def one = \x.\y.(x y);;
def two = \x.\y.(x (x y));;
...
def Pair = \x.\y.\z.(z x y);;
def Fst = \x.\y.x;;
def Snd = \x.\y.y;;

def Succ = \n.\x.\y.(x (n x y));;
def Add = \n.\m.\x.\y.(n x (m x y));;
def Mult = \n.\m.\x.(n (m x));;
```

```
def Next1 = \p.let n1 = (p Fst) in
               let n2 = (Succ (p Snd)) in
                 (Pair (Mult n1 n2) n2);;

def Nextfibo = \p.let n1 = (p Fst) in
                  let n2 = (p Snd) in
                  (Pair (Add n1 n2) n1);

def Factprim = \n.(n Next1 (Pair one zero) Fst);;

def Fiboprim = \n.(n Nextfibo (Pair zero one) Fst);;
```

The previous code cause the initial allocation of 1030 nodes which obviously cannot be erased by any computation.

In Figure 12.16 we show the maximum number of nodes allocated during the reduction of suitable applications of `Factprim` and `Fiboprim`, with and without garbage collection.

	G.C. off	G.C. on
(Factprim one I I)	1188	1178
(Factprim two I I)	1242	1219
(Factprim three I I)	1303	1276
(Factprim five I I)	1515	1383
(Factprim ten I I)	4544	1797
(Factprim (add ten five) I I)	20029	2452
(Factprim (add ten ten) I I)	73661	3226
(Fiboprim one I I)	1220	1209
(Fiboprim two I I)	1247	1234
(Fiboprim three I I)	1296	1283
(Fiboprim five I I)	1418	1381
(Fiboprim ten I I)	2439	1763
(Fiboprim (add ten five) I I)	26984	6228
(Fiboprim (add ten ten) I I)	288003	57269

Fig. 12.16. Maximum allocation space (number of nodes).

The table in Figure 12.17 shows instead, for the same terms, the

dimension of the graph at the end of the reduction, and the total number of garbage interactions performed by the garbage collector.

	G.C off	G.C. on	Garbage Op.
`(Factprim one I I)`	1128	1112	10
`(Factprim two I I)`	1146	1095	27
`(Factprim three I I)`	1199	1077	55
`(Factprim five I I)`	1454	1077	100
`(Factprim ten I I)`	4447	1077	265
`(Factprim (add ten five) I I)`	19812	1109	523
`(Factprim (add ten ten) I I)`	73274	1113	838
`(Fiboprim one I I)`	1156	1141	8
`(Fiboprim two I I)`	1144	1101	33
`(Fiboprim three I I)`	1172	1101	43
`(Fiboprim five I I)`	1302	1101	88
`(Fiboprim ten I I)`	3415	1101	623
`(Fiboprim (add ten five) I I)`	26957	1097	6117
`(Fiboprim (add ten ten) I I)`	287972	1101	66423

Fig. 12.17. Final allocation space.

The performance of the garbage collection procedure is, in some cases, surprisingly good. Consider for example the case of the λ-term `(Factprim (add ten ten) I I)`; if the garbage collector is active, the remaining nodes after the reduction are only 1113, against a final allocation of over 70000 nodes when no garbage collection is performed. Moreover, and this is particularly interesting, this result is obtained by executing only 838 garbage collecting operations! The reason is that garbage *can be duplicated along the reduction.*

Let us now consider an example involving some syntactical constructs of the extended source language. In particular, let us take a quicksort algorithm for lists of integers. Next there is the corresponding BOHM code. The function `genlist` takes an integer `n` and returns the list of the first `n` integers in inverse order, and will be used to generate the inputs to the function `quicksort`.

```
{length of a list}
def length = rec l = \x.if isnil(x)
                       then 0
```

```
                            else 1 + (l tail(x));;

{(listIt f l e) iterates f on the elements of list l}
def listIt = rec lIt =
    \f.\l.\e.if isnil(l)
                then e
                else (f head(l) (lIt f tail(l) e));;

{split a list in two sublists according the property t}
def partition = \t.\l.let switch =
    \e.\l2.if (t e) then [head(l2),cons(e,head(tail(l2)))]
                    else [cons(e,head(l2)),head(tail(l2))]
                    in (listIt switch l [nil,nil]);;

{appends two lists}
def append = rec a =
    \l1.\l2.if isnil(l1)
                then l2
                else cons(head(l1),(a tail(l1) l2));;

{generates the list of the first n integers}
def genlist = rec gen = \n.if n == 0
                              then nil
                              else cons(n,(gen n-1));;

{returns the sorted list}
def quicksort = rec qs =
  \l.if isnil(l) then nil else
    if (length l) == 1 then l else
      let l1 =
        head((partition \x.head(l) <= x tail(l))) in
      let l2 =
        head(tail((partition \y.head(l) < y tail(l)))) in
      (append (qs l1) cons(head(l),(qs l2)));;
```

The function `quicksort` has been tested on lists of several length. The results are summarized in the tables in Figure 12.18 and Figure 12.19. In particular, in Figure 12.18 we compare the maximum dimension of the graph and the number of garbage collecting interactions performed along the reduction.

	G.C. off	G.C. on	Garbage Op.
`(quicksort (genlist 5))`	3109	2542	359
`(quicksort (genlist 10))`	9442	6762	1259
`(quicksort (genlist 15))`	20627	13557	2684
`(quicksort (genlist 20))`	37912	23427	4634
`(quicksort (genlist 25))`	62547	36872	7109
`(quicksort (genlist 30))`	95782	54392	10109

Fig. 12.18. Maximum allocation space

In Figure 12.19 we compare instead the total number of interactions with and without garbage collection. As remarked above, collecting garbage as soon as possible may avoid a lot of useless operations.

	G.C. off	G.C. on
`(quicksort (genlist 5))`	5551	5393
`(quicksort (genlist 10))`	18446	17403
`(quicksort (genlist 15))`	40091	36813
`(quicksort (genlist 20))`	72111	64623
`(quicksort (genlist 25))`	116131	101833
`(quicksort (genlist 30))`	173776	149443

Fig. 12.19. Total number of interactions

12.4 Problems

BOHM works perfectly well for pure λ-calculus: much better, in average, than all "traditional" implementations. For instance, in many typical situations we have a polynomial cost of reduction against an exponential one.

Unfortunately, this is not true for "real world" functional programs. In this case, BOHM's performance is about one order of magnitude worse than typical call-by-value implementations (such as SML or Caml-light) and even slightly (but not dramatically) worse than lazy implementations such as Haskell.

The main reason of this fact should be clear: we hardly use truly

higher-order *functionals* in functional programming. Higher-order types are just used to ensure a certain degree of parametricity and polymorphism in the programs, but we never really consider functions as *data*, that is, as the real object of the computation—this makes a crucial difference with pure λ-calculus, where all data are eventually represented as functions. As a consequence, we cannot take a real advantage from the higher-order notion of sharing discussed in this book.

How much this programming style depends from the traditional, "commercial" implementations, is hard to say.

We shall not try to answer this question, here. The aim of this section is just to point out a few possible developments of BOHM that could make it really competitive with all existent reduction machines. Do not forget that BOHM is just a quite primitive prototype!

12.4.1 The case of "append"

Most of the current implementation problems can be easily understood by discussing the case of the `append` function. This is the code:

```
def append = rec append = \l1.\l2. if isnil(l1)
                    then l2
                    else cons(hd(l1), append(tl(l1) l2));;
```

Its naive translation into a graph is given in Figure 12.20.

The problem with this graph should be evident: `append` is obviously *linear* in both its arguments (the two lists to be appended). However, this linearity is hidden by the syntax, generating a graph with a lot of useless fans. This is a great source of inefficiency. For instance, in Figure 12.21 is described a typical piece of computation concerning the interaction between isnil and cons, mediated by a fan.

The current solution is to merge together (during graph generation) isnil and the fan, obtaining a new node isnil1 whose behavior is described in Figure 12.22.

Similarly, all binary forms are currently merged with fans (we leave as an easy exercise for the reader to define all new forms and their interaction rules).

The real graph generated by BOHM is thus represented in Figure 12.23.

Although this naive optimization does not work too bad in practice, the real issue of the "hidden" linearity of `append` is not solved at all. Unfortunately, this is the cause of a quite subtler problem, which is also related to BOHM's reduction strategy. As a matter of fact, graphs like

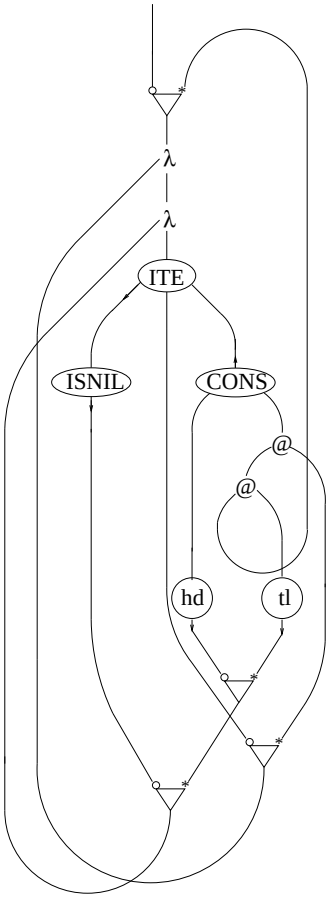


Fig. 12.20. Naive translation of append

the one in Figure 12.23 easily create situations of the kind described in Figure 12.24 (the first graph), where a safe fan is captured inside a couple of other matching fans (those inside the dotted lines).

The garbage node is blocked on the (obviously unsafe) "top-most" fan (actually, they would be merged in an $\mathsf{Uns_fan}$). Now, if the control is coming from a (i.e., this is the main path on the stack), the computation will proceed as described in Figure 12.24. In particular that the safe fan on the path leading to c would be simplified by its "interaction" with the garbage node.

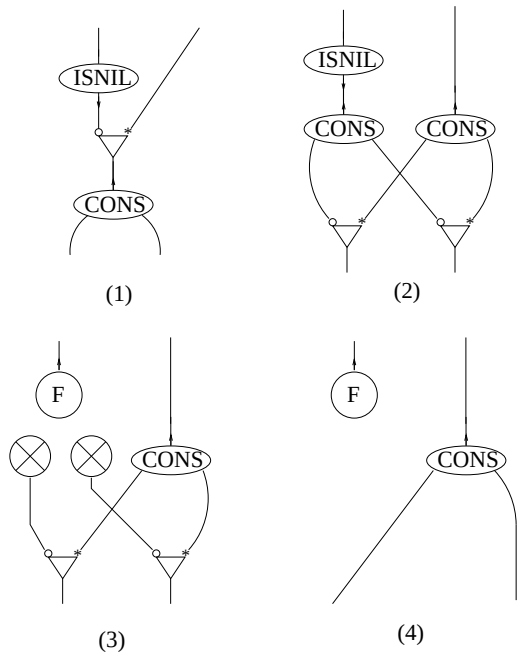


Fig. 12.21. Typical isnil-cons interaction mediated by a fan

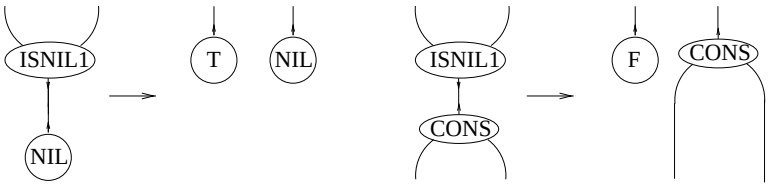


Fig. 12.22. Reduction rules for isnil1

Instead, if the control is coming from b we have no practical way to get rid of the safe fan, causing a lot of useless duplications. As a matter of fact, to solve the previous problem, we should always (or at least frequently) attempt a propagation of uns_fan nodes, that is completely infeasible, in practice.

Due to this reason, we should try to get all the possible advantage from the "hidden" linearity of programs, avoiding to introduce fans where they are not really needed.

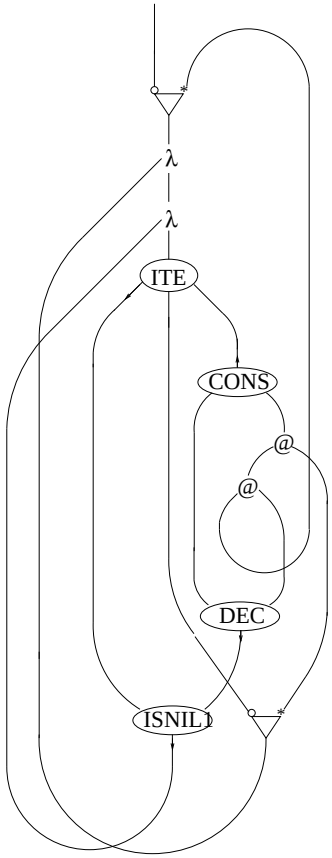


Fig. 12.23. Internal BOHM's representation of `append`

Let us see how we could proceed in the practical case of `append` (however, this is not implemented in the current version of BOHM).

First of all, we should avoid sharing between the `then` and the `else` branch of the conditional statements. This is not too difficult; it is enough to write the program in the following way:

```
def append = rec append = \l1. if isnil(l1)
               then \l2.l2
               else \l2.cons(hd(l1), append(tl(l1) l2));;
```

Such a program would generate a graph of the kind described in Figure 12.25.

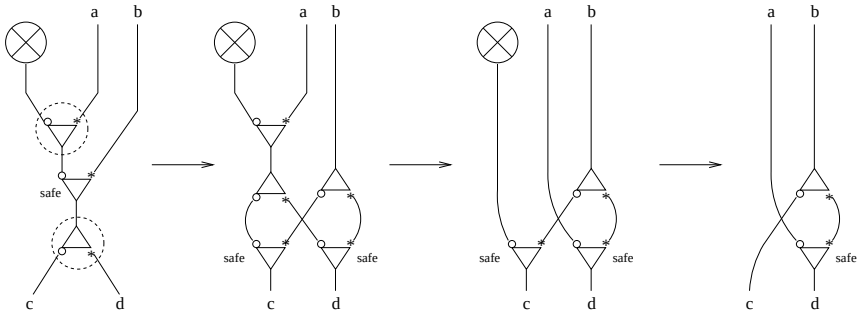


Fig. 12.24. A safe fan captured between a pair of matching fans

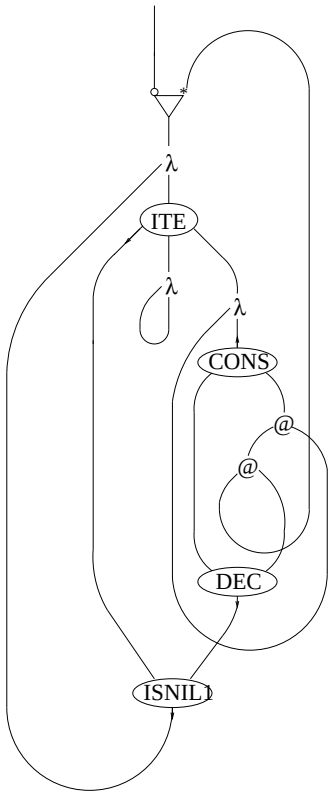


Fig. 12.25. `append` without sharing between the conditional branches

This technique can be generalized: close the subterms with respect to each (shared) variable y, and apply the result of the conditional statement to y (in the case of `append`, there is also an implicit η-reduction).

In order to eliminate the remaining sharing in the graph of Figure 12.25, we must then change the if _ then _ else statement into a case statement. The case node will pass its test-argument to the appropriate continuation(s) (so, it also works as a binder for these continuations). For instance, typical interaction rules for a simple case operator over lists are described in Figure 12.26.

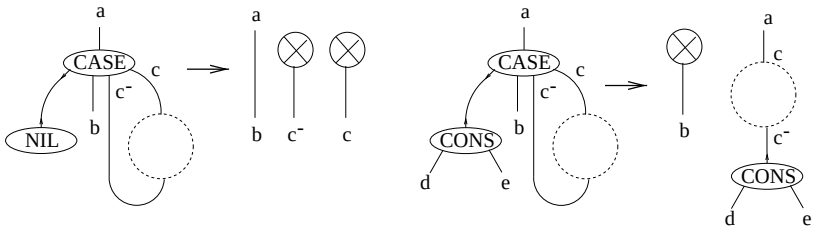


Fig. 12.26. Reduction rules for case

Using such a case node, **append** would be represented by the graph in Figure 12.27

As we expected, we have no sharing at all in this graph!

However, there is still a final problem to be addressed. If a function is linear in some of its arguments, we should not put these arguments inside a box (i.e., the arguments should be translated at the same level of the term). Note that this is *essential* if we want to profit of the linearity of the term; otherwise, we should in any case propagate a croissant through the argument, that is practically as expensive as propagating a fan. So, we must eventually address the complex problem of typing in Linear Logic.

In particular, for people acquainted with Linear Logic, it should be clear that all the previous discussion is strictly related to the so called *additive* types of this Logic. We strongly believe that the future of optimality as a really feasible architecture for functional programming largely depends on additive types and on the related linearity issues—on the other side, the *theoretical* interest of this topic is out of question.

Another interesting point to be investigated is the possibility to reintroduce, at some extent, an explicit (i.e., *global*) notion of box in the graphs. Such a box should be obviously understood as a place where we force *sequentialization* (loosing sharing). This approach seem to open the way to an integration between Lamping's algorithm and more traditional implementation techniques, such as environment machines and,

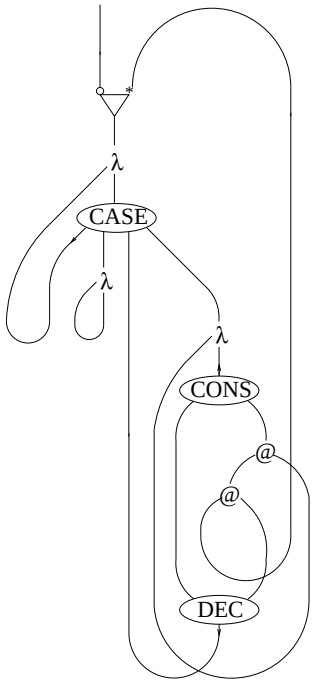


Fig. 12.27. append using case

especially, supercombinators. In particular, explicit boxes should allow a certain degree of compilation: the idea is to compile a box into the sequence of instructions needed to create an instance of itself (as it is done for supercombinators).

Bibliography

[ACCL90] Martín Abadi, Luca Cardelli, Pierre Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 31–46, San Francisco, California, January 1990.

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[Acz78] Peter Aczel. A general Church-Rosser theorem. Unpublished manuscript, 1978.

[AP81] Luigia Aiello and Gianfranco Prini. An efficient interpreter for the lambda calculus. *Journal of Computer and System Sciences*, 23:383–424, 1981.

[AKP84] Arvind, Vinod Kathail, and Keshaw Pingali. Sharing of computation in functional language implementations. Technical report, MIT, Laboratory for Computer Science, Cambridge, Massachusetts, July 1984.

[Asp92] Andrea Asperti. A categorical understanding of environment machines. *Journal of Functional Programming*, 2(1), 1992.

[Asp94] Andrea Asperti. Linear logic, comonads, and optimal reductions. *Fundamenta Informaticae*, 22(1), 1994. Special Issue devoted to Categories in Computer Science (invited paper).

[Asp95] Andrea Asperti. $\delta \circ !\epsilon = 1$: Optimizing optimal λ-calculus implementations. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95*, volume 914 of *Lecture Notes in Computer Science*, pages 102–116, Kaiserslautern, Germany, 1995. Springer-Verlag.

[Asp96] Andrea Asperti. On the complexity of beta-reduction. In *Proc. of the Twentythird Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburg Beach, Florida, 1996.

[AC96] A. Asperti, J.Chroboczek. Safe Operators: brackets closed forever. Applicable Algebra in Engineering, Communication and Computing, Springer Verlag. To appear.

[ADLR94] Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. Paths in the λ-calculus. In *Proceedings, Ninth Annual IEEE*

Symposium on Logic in Computer Science, Paris, France, 1994.

[AGN96] A. Asperti, C. Giovannetti, and A. Naletto. The Bologna Optimal Higher-order Machine. *Journal of Functional Programming*, 1996. To appear.

[AL93a] Andrea Asperti and Cosimo Laneve. Optimal Reductions in Interaction Systems. In M.-C. Gaudel and J.-P. Jounnaud, editors, *Proc. of TapSoft '93*, volume 668 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, Orsay, France, 1993.

[AL93b] Andrea Asperti and Cosimo Laneve. Paths, computations and labels in the λ-calculus. In C. Kirchner, editor, *Rewriting Techniques and Applications, Proc. of the 5th International Conference, RTA '93*, volume 690 of *Lecture Notes in Computer Science*, pages 152–167, Montreal, Canada, 1993. Springer-Verlag.

[AL94a] Andrea Asperti and Cosimo Laneve. The family relation in interaction systems. In *Proc. of the International Symposium on Theoretical Aspects of Computer Science (TACS'94)*, volume 789 of *Lecture Notes in Computer Science*, pages 366–384. Springer-Verlag, Sendai, Japan, 1994.

[AL94b] Andrea Asperti and Cosimo Laneve. Interaction Systems I: the theory of optimal reductions. *Mathematical Structures in Computer Science*, 4:457–504, 1994.

[AL95a] Andrea Asperti and Cosimo Laneve. Comparing λ-calculus translations in sharing graphs. In M. Dezani-Ciancaglini and Gordon Plotkin, editors, *Proc. of the Second International Conference on Typed Lambda Calculi and Applications, TLCA'95*, volume 902 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Edinburgh, Scotland, 1995.

[AL95b] Andrea Asperti and Cosimo Laneve. Paths, Computations and Labels in the λ-calculus. *Theoretical Computer Science*, 142(2), May 1995. Special Issue devoted to *RTA '93*. Extended version of [AL93b].

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures. An introduction to Category Theory for the Working Computer Scientist*. Foundation of Computing Series. M.I.T. Press, 1991.

[AM97] A. Asperti, H. Mairson. Parallel beta reduction is not elementary recursive. Draft. To appear.

[AJ89] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The Computer Journal*, 32(2):127–141, April 1989. Special issue on Lazy Functional Programming.

[Bar84] Henk P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1984.

[BKKS87] H.P. Barendregt, Kennaway, Klop, and Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987.

[BBdPH92] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. Term assignement for intuitionistic linear logic. Internal report, University of Cambridge, 1992.

[BL79] Gérard Berry and Jean-Jacques Lévy. Minimal and optimal computations of recursive programs. *Journal of the ACM*, 26(1):148–175, January 1979.

[BD72] Corrado Böhm and Mariangiola Dezani. A CUCH-machine: the automatic treatment of bound variables. *International Journal of Computer and Information Sciences*, 1(2):171–191, June 1972.

[BD73] Corrado Böhm and Mariangiola Dezani. Notes on "A CUCH-machine: the automatic treatment of bound variables". *International Journal of Computer and Information Sciences*, 2(2):157–160, June 1973.

[vEB90] P. van Emde Boas. *Machine models and simulation.* **Handbook of Theoretical Computer Science**, volume A, pp. 1–66. North Holland, 1990.

[BG66] Corrado Böhm and Wolf Gross. Introduction to the CUCH. In E. R. Caianiello, editor, *Automata Theory*, pages 35–65. Academic Press, 1966.

[Bou54] N. Bourbaki. *Théorie des ensembles.* Hermann & C. Editeurs, 1954.

[Bur91] A. Burroni. Higher dimensional word problems. In *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 94–105. Springer-Verlag, 1991.

[Car84] Luca Cardelli. Compiling a functional language. In *Proceedings of the ACM Symposyum on LISP and Functional Programming*, pages 208–217, Austin, August 1984.

[Carar] Luca Cardelli. The functional abstract machine. *Polymorphism*, 1(1), year?

[Chu41] A. Church. **The Calculi of Lambda-conversion**. Princeton University Press, 1941.

[CP61] A.H. Clifford and G.B. Preston. The algebraic theory of semi-groups. *A.M.S.*, 7:??, 1961.

[CCM85] G. Cousineau, P. L. Curien, and M. Mauny. The categorical abstract machine. In J.R. Jounaund, editor, *Functional Languages and Computer Architectures*, volume 201 of *LNCS*, pages 50–64. Springer-Verlag, 1985.

[Cur93] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Progress in theoretical computer science. Birkhauser, Boston, 2nd edition, 1993.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic, vol. 1.* Studies in Logic and The Foundations of Mathematics. North-Holland, Amsterdam, 1958. Third printing 1974.

[Dan90] Vincent Danos. *Une Application de la Logique Linéaire à l'Étude des Processus de Normalisation (principalement du λ-calcul).* Thèse de doctorat, Université Paris VII, 1990.

[DR93] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction. In *Proc. 8$^{\text{th}}$ Annual Symposium on Logic in Computer Science (LICS'93)*, pages 296–306, Montreal, 1993.

[DR95a] Vincent Danos and Laurent Regnier. Proof-nets and the Hilbert space. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 307–328. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[DR95b] Vincent Danos and Laurent Regnier. Reversible and irreversible computations: Goi and λ-machines. Draft, 1995.

[Fie90] John Field. On laziness and optimality in lambda interpreters: tools for specification and analysis. In *Proc. Seventeenth Symposium on Principles of Programmining Languages (POPL'90)*, pages 1–15, 1990.

[GAL92a] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26, Albequerque, New Mexico, January 1992.

[GAL92b] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. Linear logic without boxes. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science (LICS'92)*, pages 223–234, Santa Cruz, CA, June 1992. IEEE.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[Gir88] Jean-Yves Girard. Geometry of Interaction 2: Deadlock-free algorithms. In *Proc. of the International Conference on Computer Logic (COLOG 88)*. COLOG 88, Springer Verlag, 1988.

[Gir89a] Jean-Yves Girard. Geometry of Interaction 1: Interpretation of system F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloqium '88*, pages 221–260. Elsevier (North-Holland), 1989.

[Gir89b] Jean-Yves Girard. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1989. Translated and with appendices by Paul Taylor, Yves Lafont.

[Gir95a] Jean-Yves Girard. Geometry of Interaction III: The general case. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 329–389. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[Gir95b] Jean-Yves Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 1–42. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[GL87] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In *TAPSOFT '87 vol. 2*, volume 250 of *LNCS*, pages 52–66, 1987.

[GMM96] S. Guerrini, S. Martini, and A. Masini. Coherence for sharing proof nets. In Harald Ganzinger, editor, *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA-96)*, volume 1103 of *Lecture Notes in Computer Science*, pages 215–229, New Brunswick, NJ, USA, 1996. Springer-Verlag.

[Gue95] Stefano Guerrini. Sharing-morphisms and (optimal) λ-graph reductions. In ??, editor, *The Tbilisi Symposyum on Language, Logic and Computation*, ??, page ??, Tbilisi, Georgia, October 1995. CSLI Publications.

[Gue96] Stefano Guerrini. *Theoretical and Practical Issues of Optimal Implementations of Functional Languages*. PhD Thesis, Dipartimento di Informatica, Università di Pisa, Pisa, 1996. TD-3/96.

[Gue97] Stefano Guerrini. A general theory of sharing graphs. IRCS Report 97–04, Institute for Reasearch in Cognitive Science, University of Pennsylvania, Philadelphia, March 1997. Submitted for publication: invited submission to a special issue of Theoretical Computer Science.

[HS86] J. Roger Hindley and Jonathan Paul Seldin. *Introduction to Combinators and λ-calculus*, volume 1 of *London Mathematical Society, Student Texts*. Cambridge University Press, Cambridge, 1986.

[HU79] J. E. Hopcroft and J. D. Ullman. **Introduction to Automata**

Theory, Languages, and Computation. Addison Wesley, 1979.

[Kat90] Vinod Kathail. *Optimal Interpreters for lambda-calculus based functional languages.* PhD Thesis, MIT, 1990.

[Klo80] J. W. Klop. *Combinatory Reduction Systems.* PhD Thesis, Matematisch Centrum, Amsterdam, 1980. Mathematical Centre Tracts 127.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988. Special issue: Int. Joint Conf. on Theory and Practice of Software Development, Pisa, March 1987.

[Laf90] Yves Lafont. Interaction nets. In *Proc. of Seventeenth Annual ACM Symposyum on Principles of Programming Languages (POPL'90)*, pages 95–108, San Francisco, California, January 1990.

[Laf92] Yves Lafont. Penrose diagrams and 2-dimensional rewritings. In *LMS Symposium on Applications of Categories in Computer Science.* Cambridge University Press, 1992.

[Laf95] Yves Lafont. From proof nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 225–247. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[Lam89] John Lamping. An algorithm for optimal lambda calculus evaluation. Technical Report Series SSL-89-27, Xerox PARC, Palo Alto, May 1989.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. of Seventeenth Annual ACM Symposyum on Principles of Programming Languages*, pages 16–30, San Francisco, California, January 1990.

[Lan63] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1963.

[Lan93] Cosimo Laneve. *Optimality and Concurrency in Interaction Systems.* PhD Thesis, TD-8/93, Dipartimento di Informatica, Università di Pisa, Pisa, March 1993.

[LM96] Julia L. Lawall and Harry G. Mairson. *Optimality and inefficiency: what isn't a cost model of the lambda calculus?* **1996 ACM International Conference on Functional Programming**, pp. 92–101.

[LM97] Julia L. Lawall and Harry G. Mairson. *On the global dynamics of optimal graph reduction.* **1997 ACM International Conference on Functional Programming**, to appear.

[LM92] X. Leroy and M. Mauny. The Caml Light system, release 0.5. documentation and user's manual. Technical report, INRIA, September 1992.

[Lév76] Jean-Jacques Lévy. An algebraic interpretation of the λβK-calculus and an application of labelled λ-calculus. *Theoretical Computer Science*, 2(1):97–114, 1976.

[Lév78] Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le lambda-calcul.* PhD Thesis, Université Paris VII, 1978.

[Lév80] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In Seldin and Hindley [SH80], pages 159–191.

[Mac95] Ian Mackie. The Geometry of Interaction Machine. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–208, San Francisco, California, January 1995.

[Mai92] Harry G. Mairson. *A simple proof of a theorem of Statman.* **Theoretical Computer Science** 103 (1992), pp. 387–394.

[Mey74] Albert R. Meyer. *The inherent computational complexity of theories of ordered sets.* **Proceedings of the International Congress of Mathematicians**, 1974, pp. 477–482.

[Par92] Michel Parigot. λμ-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning, International Conference LPAR'92*, volume 624 of *Lecture Notes in Computer Scienc*, St. Petersburg, Russia, 1992. Springer-Verlag.

[Pet84] Mario Petrich. *Inverse Semigroups.* Pure and applied mathematics. John Wiley & Sons, New York, 1984.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International, Englewood Cliffs, New Jersey, 1987.

[Reg92] Laurent Regnier. *Lambda-Calcul et Reseaux.* Phd Thesis, Université Paris 7, January 1992.

[Sch82] Helmut Schwichtenberg. *Complexity of normalization in the pure typed lambda calculus. The L.E.J. Brouwer Centenary Symposium* (ed. A. S. Troelstra and D. Van Dalen). North-Holland, Amsterdam. 1982.

[SH80] Jonathan P. Seldin and J. Roger Hindley, editors. *To H.B. Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[Sta79] Richard Statman. *The typed λ-calculus is not elementary recursive.* **Theoretical Computer Science** 9, 1979, pp. 73–81.

[The94] The Yale Haskell Group. The yale haskell users manual. Technical report, Yale University, October 1994.

[VDS94] J.B. Joinet V. Danos and H. Shellinx. lkt and lkq. Submitted for publication, 1994.

[Vui74] J. Vuillemin. Correct and optimal implementation of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3), 1974.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the lambda-calculus.* Phd Thesis, Oxford, 1971. Chapter 4.

All traditional implementation techniques for functional languages (mostly based on supercombinators, environments or continuations) fail to avoid useless repetition of work; they are not 'optimal' in their implementation of sharing, often causing a catastrophic, exponential explosion in reduction time. Optimal reduction is an innovative graph reduction technique for functional expressions, introduced by Lamping in 1990, that solves the sharing problem. This book, the first in the subject, is a comprehensive account by two of its leading exponents. Practical implementation aspects are fully covered as are the mathematical underpinnings of the subject. The relationship to the pioneering work of Lévy and to Girard's more recent *Geometry of Interaction* are explored; optimal reduction is thereby revealed as a prime example of how a beautiful mathematical theory can lead to practical benefit.

The book is essentially self-contained, requiring no more than basic familiarity with functional languages. It will be welcomed by graduate students and research workers in lambda calculus, functional programming or linear logic.

Cambridge Tracts in Theoretical Computer Science

CAMBRIDGE
UNIVERSITY PRESS

ISBN 0-521-62112-7

9 780521 621120