Robert Hirschfeld Kim Rose (Eds.)

Self-Sustaining Systems

First Workshop, S3 2008
Potsdam, Germany, May 15-16, 2008
Revised Selected Papers

Springer

Volume Editors

Robert Hirschfeld
Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Straße 2-3, 14482 Potsdam, Germany
E-mail: hirschfeld@hpi.uni-potsdam.de

Kim Rose
Viewpoints Research Institute
1209 Grand Central Avenue, Glendale, CA 91201, USA
E-mail: kim.rose@vpri.org

Library of Congress Control Number: 2008938339

CR Subject Classification (1998): D.2, F.1.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-89274-5 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-89274-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12511341 06/3180 5 4 3 2 1 0

Preface

The Workshop on Self-sustaining Systems (S3) is a forum for the discussion of topics relating to computer systems and languages that are able to bootstrap, implement, modify, and maintain themselves. One property of these systems is that their implementation is based on small but powerful abstractions; examples include (amongst others) Squeak/Smalltalk, COLA, Klein/Self, PyPy/Python, Rubinius/Ruby, and Lisp. Such systems are the engines of their own replacement, giving researchers and developers great power to experiment with, and explore future directions from within, their own small language kernels.

S3 took place on May 15–16, 2008 at the Hasso-Plattner-Institute (HPI) in Potsdam, Germany. It was an exciting opportunity for researchers and practitioners interested in self-sustaining systems to meet and share their knowledge, experience, and ideas for future research and development. S3 provided an opportunity for a community to gather and discuss the need for self-sustainability in software systems, and to share and explore thoughts on why such systems are needed and how they can be created and deployed. Analogies were made, for example, with evolutionary cycles, and with urban design and the subsequent inevitable socially-driven change.

The S3 participants left with a greater sense of community and an enthusiasm for probing more deeply into this subject. We see the need for self-sustaining systems becoming critical not only to the developer's community, but to end-users in business, academia, learning and play, and so we hope that this S3 workshop will become the first of many.

We would like to thank our invited speakers for their insightful and provocative talks, our presenters for their technical contributions, our members of the program committee for their constructive reviews, all participants for their interest, and the local organizers for their exemplary support.

June 2008

Robert Hirschfeld
Kim Rose

Organization

The Workshop on Self-sustaining Systems (S3) 2008 was organized by the Software Architecture Group of the Hasso-Plattner-Institute (HPI) at the University of Potsdam, Germany, and the Viewpoints Research Institute (VPRI), California, USA.

Chairs

Robert Hirschfeld	Hasso-Plattner-Institut, Germany
Kim Rose	Viewpoints Research Institute, USA

Program Committee

Johan Brichau	Université Catholique de Louvain, Belgium
Pascal Costanza	Vrije Universiteit Brussel, Belgium
Wolfgang De Meuter	Vrije Universiteit Brussel, Belgium
Stéphane Ducasse	INRIA Lille, France
Richard P. Gabriel	IBM Research, USA
Michael Haupt	Hasso-Plattner-Institut, Germany
Robert Hirschfeld	Hasso-Plattner-Institut, Germany
Dan Ingalls	Sun Microsystems Laboratories, USA
Martin von Löwis	Hasso-Plattner-Institut, Germany
Hidehiko Masuhara	University of Tokyo, Japan
Ian Piumarta	Viewpoints Research Institute, USA
David Ungar	IBM, USA

Local Organization

Malte Appeltauer	Hasso-Plattner-Institut, Germany
Michael Haupt	Hasso-Plattner-Institut, Germany
Robert Krahn	Hasso-Plattner-Institut, Germany
Jens Lincke	Hasso-Plattner-Institut, Germany
Michael Perscheid	Hasso-Plattner-Institut, Germany
David Tibbe	Hasso-Plattner-Institut, Germany
Sabine Wagner	Hasso-Plattner-Institut, Germany

Sponsoring Institutions

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Viewpoints Research Institute
1209 Grand Central Avenue Glendale, CA 91201 tel. (818) 332-3001 fax (818) 244-9761

Table of Contents

Open, Extensible Object Models

Ian Piumarta and Alessandro Warth

Viewpoints Research Institute
1209 Grand Central Avenue
Glendale, CA 91201, USA
ian@vpri.org

Abstract. Programming languages often hide their implementation at a level of abstraction that is inaccessible to programmers. Decisions and tradeoffs made by the language designer at this level (single vs. multiple inheritance, mixins vs. Traits, dynamic dispatch vs. static case analysis, etc.) cannot be repaired easily by the programmer when they prove inconvenient or inadequate. The artificial distinction between implementation language and end-user language can be eliminated by implementing the language using only end-user objects and messages, making the implementation accessible for arbitrary modification by programmers. We show that three object types and five methods are sufficient to bootstrap an extensible object model and messaging semantics that are described entirely in terms of those same objects and messages. Raising the implementation to the programmers' level lets them design and control their own implementation mechanisms in which to express concise solutions and frees the original language designer from ever having to say "I'm sorry".

1 Introduction

Most programming languages and systems make a clear distinction between the *implementation level* in which the system is built and the *'end-user' level* in which programs are subsequently written. The abstractions and semantics provided by these programming systems are effectively immutable. Metaobject Protocols (MOPs) [5] are designed to give back some power to programmers, letting them extend the system with new abstractions and semantics. We are interested in a different approach to solving the same problem where we eliminate the distinction between the implementation and user levels of the programming system.

As an example of the problem we are trying to solve, consider the implementation of a Lisp-like language with several atomic object types. The implementer must choose a representation for these objects in some (typically lower-level) implementation language. The choice of representation can have a profoundly limiting effect on the ability of both the implementer and end-user to extend the language with new types, primitive functionality and semantics at some later time. Our Lisp-like end-user language might have C as its implementation language and use a *discriminated union* to store atomic objects and 'cons' cells:

```
enum ObjectTag { Number, String, Symbol, Cons };
struct Object {
  enum ObjectTag  tag;
  union {
    struct Number number;
    struct String string;
    struct Symbol symbol;
    struct Cons   cons;
  } payload;
};
```

With this representation, each primitive in the end user language that manipulates data would use conditional (`if` or `switch`) statements to select appropriate behaviour according to the `tag` field.

This simple object model has already made significant design decisions and rendered them immutable:

– All objects must start with an integer `tag` field.
– The internal layout of the four intrinsic types cannot be modified at runtime.

The consequences of these decisions include:

– New payloads cannot be added by end user code, especially if they require more storage than the intrinsic types.
– New tags cannot be added unless all primitives are explicitly designed to work in the presence of arbitrary tags, or the user is in a position to understand, modify and then recompile every part of the base language implementation that might be concerned with object tags.

We could start to address these problems by creating a more general object model for our structured data, for example by adding a `size` field to allow for arbitrary payloads. Unfortunately each such change *adds* complexity to the language runtime and imposes *more* 'meta-structure' in the objects, ultimately making them *less* amenable to unanticipated deep modifications in the future.

These problems are even more severe when we consider object-oriented languages. The object model for a simple prototype-based language might specify 'method dictionary' and 'parent' slots in every object. The runtime would look up a message name in the receiver's `methodDictionary`, trying again in the `parent` object's method dictionary if no match is found, continuing until finding a match or reaching the end of the parent chain. Adding multiple delegation to this language would be difficult because the runtime assumes that the `parent` field contains a single object and not, for example, a list of parent objects to try in turn.

The trouble is that some of the semantics of the above example (single delegation between instances) are reified eagerly in the execution mechanisms of the language. This in turn eagerly imposes supporting meta-structure (instances chained through a `parent` slot) within the objects. Since the execution mechanisms are expressed in an implementation language at a lower level of abstraction than that of the end user language, neither the mechanisms nor their effects on object structure can be modified by end users. Moreover, adapting the implementation machinery for reuse in supporting a different end-user language is more difficult when the required changes are pervasive and expressed in a low-level implementation language. In this paper we present an object model intended to eliminates most of these problems:

– We show how an object-based model of data can help alleviate some of the problems of extensibility in programming language implementation (Section 2).
– We define a simple, extensible object model that imposes no structure on end-user objects (Section 3).
– The end-user object model provides message-passing semantics implemented using its own objects and messaging mechanism, making the semantics of messaging modifiable or even replaceable from within the end-user language. We show that three kinds of object and five small methods are sufficient to achieve this (Section 3.1).
– The flexibility gained by exposing the object model's semantics is illustrated by showing that it can be extended easily to support language features including multiple inheritance and mixed-mode execution [10] (Sections 2.2 and 3).
– We validate the use of this approach for production systems by showing that: it has low space overhead (Section 5); its performance can be competitive with, and in some cases even better than, equivalent 'static' implementation techniques (Section 5.3); existing object models can be easily implemented on top of the model (Section 5.1); advanced compositional techniques such as Traits [11] can be accommodated (Section 5.2).

2 The Object Model by Example

The object model describes one thing: how an object responds to a message. Each object is associated with a *vtable* object. When a message is sent to an object O, its vtable V is asked to find the appropriate method to run. This is done by sending the message '`lookup`' to V, with the message name as argument. The semantics of sending a message to O are therefore determined entirely by V's response to the '`lookup`' message. By overriding (or redefining) '`lookup`' we can change the semantics of message sending for some (or all) objects.

The vtable object doesn't have to be a table. It can determine the method to run for a given message send in any way it wants. Often, though, vtables are simply dictionaries mapping message names onto method implementations.

This section introduces the object model by using it to solve two of the problems mentioned in the introduction: adding a new atomic object type to a Lisp-like language and converting single delegation to multiple delegation in a message-passing language.

2.1 Adding Data Types to a Language

For our Lisp-like language we might have a `length` primitive that tells us how many elements are present in a string or list. Using the `tag` field in the `Object` structure to discriminate the type of payload, `length` might look like this:

```
int length(struct Object *object)
{
  switch (object->tag)
  {
    case Number: error("numbers have no length");
    case String: return object->payload.string.length;
    case Symbol: error("symbols have no length");
    case Cons:   return object->payload.cons.cdr
               ? 1 + length(object->payload.cons.cdr)
               : 1;
    default:     error("illegal tag");
  }
}
```

Let's add a vector type to this language. We have to extend the above `switch` statement with a new `case` to take into account our new data type and its `tag` value:

```
case Vector: return object->payload.vector.length;
```

This isn't too bad if we are the only user of the language and we have access to the source code of the implementation. However, the situation is much worse if we want to share the new type with other users of the language, possibly as a third-party extension; any primitive that is not modified with an additional `case` to handle vectors will cause a run-time error.

It would be better to store the relevant `case` implementation from each primitive function in the data type itself. Using our object model the new data type is added to the language by creating a new vtable (object behaviour) and then installing its primitives as methods in the vtable. Figure 1 shows what this would look like in our object model, again using C as the implementation language.

This is more than advocating an object-oriented style of programming language construction. Consider the same Lisp-like language implemented in C++. Even if the `length` primitive was made a virtual function of each supported data type, we would have to recompile every file after adding `Vector` since the layout of C++ vtables is computed at compile time; adding a new virtual method would invalidate all previous assumptions about the vtable layout.

```
struct vtable *Vector_vt = 0;

int Vector_length(struct Vector *vector) {
  return vector->length;
}

void initialise(void) {
  ...
  Vector_vt = send(vtable, s_allocate,
                   sizeof(struct vtable));
  send(Vector_vt, s_addMethod, s_length, Vector_length);
  ...
}

int length(struct object *object) {
  return send(object, s_length);
}
```

Fig. 1. Creating a new type and associating functionality with it. The vtable `Vector_vt` describes the behaviour of the new type. Invoking the method `s_addMethod` in it makes an association between the selector `s_length` and the method implementation `Vector_length`. The `length` primitive can now simply invoke the method `s_length` in any object and expect it to respond appropriately regardless of the number of data types supported by—or added to—the language. (The variables prefixed with `s_` are symbols: interned, unique strings suitable for identifying method names.)

Perhaps more compelling is an example involving an object-oriented language that uses the object model and that can directly modify the semantics of its own messaging mechanism.

2.2 Adding Multiple Inheritance to a Prototype-Based Language

This example uses a high-level, prototype-based programming language with single delegation that uses the object model directly for its end user objects.[1] We will use this language for several examples. Its syntax is very close to that of Smalltalk [4] with a few small differences (described in Appendix A).

Everything in our object model is an object, including the vtables that describe the behaviour of objects. Interacting with vtables is just a matter of invoking methods in them. One such method is called `lookup`; it takes a method name as an argument and returns a corresponding method implementation. By overriding (or redefining) this method we can change the semantics of message sending for some (or all) objects.

The prototype-based language provides the programmer with single inheritance; a given *family* of objects *inherits* behaviour from a parent family (with

[1] This language is written entirely in itself and can be downloaded, along with many examples including those presented in this paper, from
`http://piumarta.com/software/cola`

```
ParentList : List ()

vtable addParent: aVtable
[
  parent isNil
    ifTrue: [parent := aVtable]
    ifFalse:
     [parent isParentList
        ifTrue:  [parent add: aVtable]
        ifFalse: [parent := ParentList new
                    add: parent;
                    add: aVtable;
                    yourself]]
]

ParentList lookup: messageName
[
  | method |
  self do: [:aVtable |
    (method := aVtable lookup: messageName) notNil
      ifTrue: [↑method]].
  ↑nil
]
```

Fig. 2. Adding multiple inheritance to a prototype-based language. We will store multiple parents in `ParentList` objects; these extend (inherit behaviour from) `List` without adding any additional state. We tell `vtable` how to `addParent:` by converting a single parent `vtable` into a `ParentList` if necessary, then `add`ing the new parent `vtable` to the list. Next we define `lookup:` for `ParentList` to search for the `messageName` in each parent consecutively. (The `lookup:` method already installed in `vtable` can be left in place; it performs a depth-first search up the inheritance chain by invoking `lookup:` in its `parent` slot, which can now be either a `vtable` or a `ParentList`.)

all families eventually inheriting behaviour from `Object`). Figure 2 shows how the programmer can directly add multiple inheritance to this language, without loss of performance.[2] With these additions to the language, and given three prototype families `C1`, `C2` and `C3`

```
C1 : Object ()
C1 m  [ 'this is m' putln ]

C2 : Object()
C2 n  [ 'this is n' putln ]

C3 : C1 ()  "C3 inherits from C1"
```

[2] The message sending mechanism uses a *method cache* to memoize the result of invoking `lookup` in a given `vtable` for a given `messageName`. The overhead of iterating through multiple parents is incurred only when the method cache misses, which is rarely [2].

the programmer can now dynamically add `C2` as a parent of `C3`

```
C3 vtable addParent: C2 vtable
```

so that objects in its family can execute methods inherited from both `C1` and `C2`:

```
C3 new
  m;  "inherited from C1"
  n   "inherited from C2"
```

A serious implementation would of course have to take state and behavioural conflicts into account, although this could be as simple as allowing only one parent to be stateful and disallowing duplicated message names. (Our implementation of Traits [11] in Section 5.2 illustrates this.)

3 Open, Extensible Object Models

An object typically describes both *state* and *behaviour* that acts on (or is influenced by) that state. We might account for both state and behaviour in the object model, but it would be simpler to model just one of them and then use it to provide the other indirectly. We choose to model (and expose) behaviour as a set of *methods* that are invoked in an object by name; access to state, if appropriate, is then provided through 'accessor' methods.[3]

Figure 3 illustrates this simple model: an object is some opaque quantity in which a method can be invoked by name; we call the set of methods associated with a given object its *behaviour*. Since we wish to avoid imposing structure on end user objects, the description of behaviour is stored separately from the object in a manner similar to most object-oriented languages; in particular, `parent` slots and method tables are not stored in objects. An object is therefore a *tuple* of behaviour and state. Since the behaviour is decoupled from the internal state of the object it can be replaced and/or shared as desired, or even associated implicitly with the object.[4]

Figure 4 shows the layout of objects in memory. An ordinary object pointer (oop) points to the first byte of the object's internal state (if any). The object's behaviour is described by a *virtual table* (vtable). A pointer to the vtable is placed immediately before the object's state, at offset -1 relative to the pointer. This is done to preserve pointer identity for objects that encapsulate a foreign structure, facilitating communication with the operating system and libraries. It also allows compiled methods, identified by the address of their first instruction, to be full-fledged objects.

A vtable is an object too, as shown in Figure 5, and has a reference to the 'vtable for vtables' before its internal state. This 'vtable for vtables' is its own

[3] The discussion of related work (Section 6) mentions Self, a system that made the opposite choice of modeling behaviour as a special kind of state.

[4] In the prototype language, tagged (odd) pointers and the null pointer are implicitly associated with vtables for the behaviour of small integers and nil, respectively.

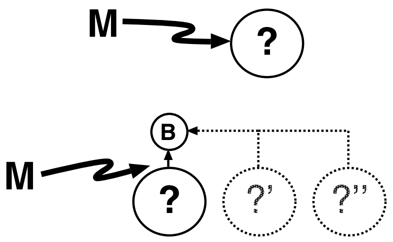


Fig. 3. Minimal object model. An object is some opaque state *?* on which a method *M* can be invoked by name. To implement this model we need a mapping from method names to method implementations. So, to invoke a method *M* in the object *?* we find the corresponding method implementation in a behaviour description *B*. An object is therefore a tuple of behaviour *B* and state *?*. Since behaviour is separate from the object it describes, it is possible to share any given behaviour *B* between several distinct objects *?*, *?'*, *?''*, . . .

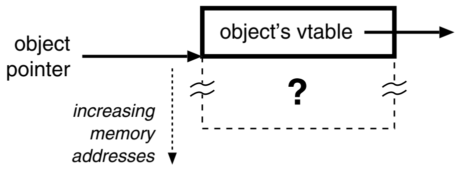


Fig. 4. Implementation of minimal object. An object pointer (oop) points to the start of the object's internal state (if any). The object's behaviour is described by a *virtual table* (vtable). A pointer to the vtable is placed one word before the object's state.

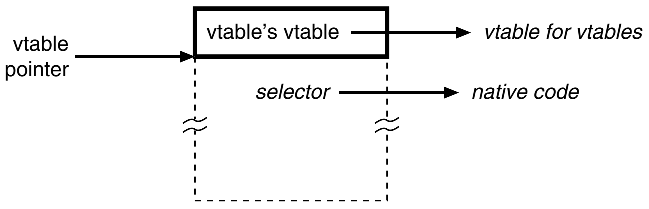


Fig. 5. Internals of vtables. A vtable maps a message name (selector) onto the address of the native code that implements the corresponding method. The mapping is determined by the vtable's response to the `lookup` message, which is bound to an implementation by the 'vtable for vtables'.

vtable, as shown in Figure 6. It provides a default implementation of the `lookup` method (for all vtables) that maps message names onto method implementations. The state within a vtable supports this mapping. The `lookup` method therefore dictates the internal structure of all vtables, but there is nothing special about the initial 'vtable vtable' nor the structure of vtables; a new 'vtable vtable' can

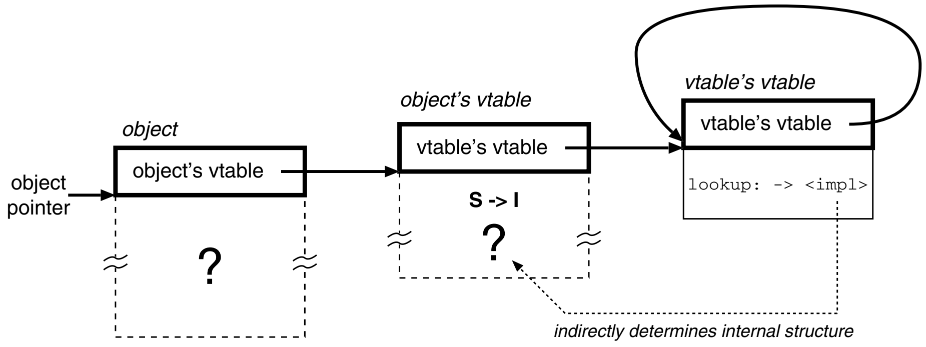


Fig. 6. Everything is an object. Every object has a vtable that describes its behaviour. A method is looked up in a vtable by invoking its `lookup` method. Hence there is a 'vtable vtable' that provides an implementation of `lookup` for all vtables in the system, including for itself. The implementation of this `lookup` method is the only thing in the object model that imposes internal structure on vtables.

Table 1. Essential objects and methods. For `vtables`, `addMethod` creates an association from a message name to a method implementation, `lookup` queries the associations to find an implementation corresponding to a message name, `delegated` creates a new `vtable` that will delegate unhandled messages to the receiver, and `allocate` creates a new object within the `vtable`'s family (by copying the receiver into the new object's `vtable` slot). We include `symbol`'s `intern` method in this list since the end user must have some way to (re)construct the name of a method. The vtables for `vtable` and `symbol` delegate to the vtable for `object`, to ease the creation of singly-rooted hierarchies in which these types are reused directly as end-user object types.

type	*method*
object	
symbol	intern
vtable	addMethod
vtable	lookup
vtable	allocate
vtable	delegated

be created at any time to provide a new `lookup` method that implements a family of vtables with arbitrarily different semantics and internal structure.[5]

3.1 Essential Objects and Methods

Table 1 lists the three essential object types and the five essential methods that they implement. These methods are described below, with implementations

[5] The method `addMethod`, described below, also depends on the internal structure of vtables and would be overridden in parallel with the `lookup` method when changing their structure.

```
let SymbolList = EmptyList

function symbol_intern(self, string) =
  foreach symbol in SymbolList
    if string = symbol.string
      return symbol
  let symbol = new symbol(string)
  append(SymbolList, symbol)
  return symbol
```

Fig. 7. Method symbol.intern. Symbols are unique strings. A lazy implementer would co-opt a `vtable` into use as a `SymbolList` holding previously-interned `symbols`.

```
function vtable_addMethod(self, symbol, method) =
  foreach i in 1 .. self.size
    if self.keys[i] = symbol
      self.values[i] := method
      return
  append(self.keys, symbol)
  append(self.values, method)
```

Fig. 8. Method vtable.addMethod. If the method name `symbol` is already present, replace the method associated with it. Otherwise add a new association between the name and the method.

```
function vtable_lookup(self, symbol) =
  foreach i in 1 .. self.size
    if self.keys[i] = symbol
      return self.values[i]
  if self.parent ≠nil
    return self.parent.lookup(symbol)
  return nil
```

Fig. 9. Method vtable.lookup. The default implementation searches the receiver's keys for the message name. If no match is found the search continues in the parent, if present, otherwise the search fails by answering `nil`.

shown in pseudo-code intended to make their operation as clear as possible. (Appendix B presents a complete implementation of these methods and types in GNU C.)

Before we can construct an object system we need a way to add methods to `vtables`, which requires a means to construct unique method names. Figure 7 shows a simple algorithm for creating 'interned' (unique) strings that are ideal for use as method names.

To add methods to a `vtable` we send `addMethod` to it, passing a message name (`symbol`) and the address of native code implementing the method. The algorithm is shown in Figure 8.

```
function vtable_allocate(self, size) =
  let object = allocateMemory(PointerSize + size)
  object := object + PointerSize
  object[-1] := self  /* vtable */
  return object
```

Fig. 10. Method vtable.allocate. A new object is created and its `vtable` (stored in the word preceding the object) is set to the `vtable` in which the `allocate` method was invoked, making the object a member of that `vtable`'s family. The `size` argument specifies the size of the object's state. Computation of the correct value for `size` is dependent on the programming language implementation in which the object model is being used.

```
function vtable_delegated(self) =
  let child =
    if self ≠nil
      vtable_allocate(self[-1], VtableSize)
    else
      vtable_allocate(nil, VtableSize)
  child.parent := self
  child.keys := EmptyList
  child.value := EmptyList
  return child
```

Fig. 11. Method vtable.delegated. A new `vtable` is `allocate`d and its `parent` set to the `vtable` in which the `delegated` method is being invoked. These `parent` fields link the `vtable`s together into a single delegation chain.

Sending a message to an object begins by mapping a particular combination of object and message into an appropriate method implementation. Figure 9 shows the algorithm for `vtable`'s `lookup` method that performs this mapping.

Invoking the `allocate` method in a `vtable` allocates a new object. The object is made a member of the vtable's family, as shown in Figure 10.

Finally, the creation of new behaviours is provided by `vtable`'s `delegated` method. It creates a new (empty) `vtable` whose parent is the vtable in which `delegated` was invoked. The algorithm is shown in Figure 11.

3.2 Message Sending

To send a message M to an object O we look up M in the vtable of O to yield a method implementation that is then called. The call passes the object O (which becomes `self` in the called method) and any remaining message arguments. The send algorithm is therefore:

```
function send(object, messageName, args...) =
  let method = bind(object, messageName)
  return method(object, args...)
```

The function bind is responsible for looking up the method name in the vtable of object and just invokes `lookup` in the object's vtable, passing messageName as the argument:

```
function bind(object, messageName) =
  let vt = object[-1]
  let method =
    if messageName = lookup
         and object = VtableVT
      vtable_lookup(vt, lookup)
    else
      send(vt, lookup, messageName)
  return method
```

Note that the recursion implied by `send` calling `bind` which in turn calls `send` (to invoke the `lookup` method in the object's vtable) is broken by 'short-circuiting' the send (calling the method `vtable_lookup` directly) when the method name is `lookup` and the object in which it is being bound is the 'vtable vtable'.

3.3 Bootstrapping the Object Universe

The structure associated with the three essential types has to be created and their vtables populated before the object model will behave as we have described. Figure 12 shows one possible order in which this initialisation can take place:

1. The vtables for `vtable`, `object` and `symbol` are created and initialised explicitly.
2. The symbol `lookup` is interned and the method `vtable.lookup` installed. At this point the `send` and `bind` functions described in the previous section (i.e., message sending) will work.
3. The symbol `addMethod` is interned and the method `vtable.addMethod` installed. At this point methods can be installed in a vtable by sending `addMethod` to the vtable.
4. The symbol `allocate` is interned and the method `vtable.allocate` installed. At this point new members of an object family can be created by sending their `vtable` the message `allocate`, and this is done to create the prototype `symbol` object.
5. The symbol `intern` is interned and the method `symbol.intern` installed. At this point new symbols can be interned by sending `intern` to the prototype `symbol` object.
6. Finally, the symbol `delegated` is interned (by sending `intern` to `symbol`) and the method `vtable.delegated` installed (by sending `addMethod` to the vtable for vtables). At this point the object system behaves exactly as described in this paper.

The initialised 'object universe' is shown in Figure 13.

```
function initialise() =
  /* 1. create and initialise vtables */
  VtableVT := vtable_delegated(nil)
  VtableVT[-1] := VtableVT

  ObjectVT := vtable_delegated(nil)
  ObjectVT[-1] := VtableVT
  VtableVT.parent := ObjectVT

  SymbolVT := vtable_delegated(ObjectVT)

  /* 2. install vtable.lookup */
  lookup := symbol_intern(nil, "lookup")
  vtable_addMethod(VtableVT, lookup, vtable_lookup)

  /* 3. install vtable.addMethod */
  addMethod := symbol_intern(nil, "addMethod")
  vtable_addMethod(VtableVT, addMethod,
                   vtable_addMethod)

  /* 4. install vtable.allocate */
  allocate := symbol_intern(nil, "allocate")
  VtableVT.addMethod(allocate, vtable_allocate)
  symbol := SymbolVT.allocate(SymbolSize)

  /* 5. install symbol.intern */
  intern := symbol_intern(nil, "intern")
  SymbolVT.addMethod(intern, symbol_intern)

  /* 6. install vtable.delegated */
  delegated := symbol.intern("delegated")
  VtableVT.addMethod(delegated, vtable_delegated)
```

Fig. 12. Bootstrapping the object model. Method implementations are called as functions and vtable slots initialised explicitly to create the vtables for the three objects types. The methods `symbol_intern` and `vtable_addMethod` are called explicitly to populate the vtables. By the time the last two lines are reached, we have enough of the object model in place that we can send messages to intern the symbol `delegated` and install it in the vtable for vtables.

3.4 Implementation Language Bindings

To deploy the object model as part of a programming language implementation, we need three things:

– Implementation language structure definitions for the layouts of `object`, `symbol` and `vtable` (implied by the default implementation of the `lookup` method installed in the 'vtable vtable');
– Implementations of the five essential methods in the implementation language; and
– An implementation language method invocation mechanism, to call a method implementation (returned from `lookup`) passing the receiver object and message arguments.

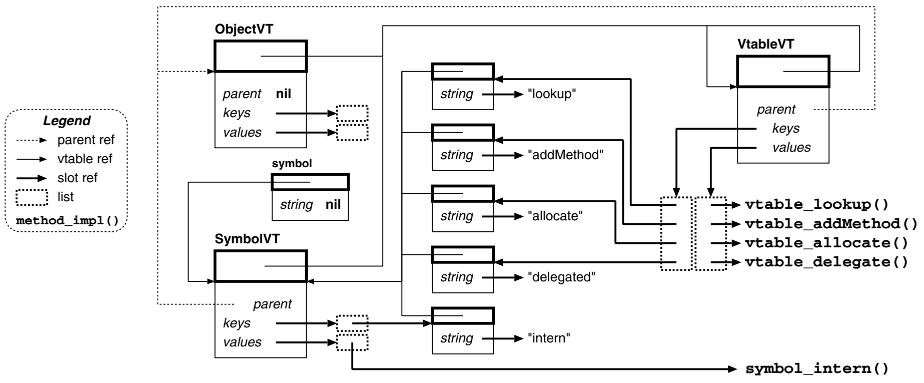


Fig. 13. The object model universe. The larger objects are the vtables for the three essential types (`object`, `symbol` and `vtable`). Just above `SymbolVT` is the prototype `symbol` object, and to the right of it are the symbols that provide message names for the five essential methods whose implementations are just below `VtableVT` on the right. The symbol `intern` is bound to the method `string_intern` in the `SymbolVT` and the remaining methods are bound to their message names in `VtableVT`. Both `SymbolVT` and `VtableVT` delegate to `ObjectVT`.

Appendix B presents a complete implementation in the GNU C language. The next section discusses two optimisations appearing in this implementation that significantly improve the performance of message sending. It should be straightforward to adapt them to other programming languages.

3.5 Optimising Performance

The performance of the GNU C versions of `send()` and `bind()` are improved by two forms of caching.

Figure 14 shows a version of `send` that is implemented as a macro. This allows each send site to remember the previous destination method returned by `bind` in an *inline cache*. As long as the vtable of the next receiver does not change, the previous destination method can be invoked directly without calling `bind` again (assuming the message name at the send site is constant).

Figure 15 shows a version of `bind` that has been optimised with a global *method cache*. Before invoking `lookup` the optimised `bind` looks for the vtable and method name in a cache of previously bound methods. If it finds a match, it returns the cached closure; if not, it invokes `lookup` and fills the appropriate cache line.

These two optimisations are independent and can be used separately or together. Note that a realistic language implementation would need a way to invalidate these caches each time a change is made to vtable contents or inheritance relationships. Mechanisms for doing this are simple but beyond the scope of this paper.

```
#define send(OBJ, MSG, ARGS...) ({                         \
         struct object *o = (struct object *)(OBJ); \
         struct vtable *thisVT = o->_vt[-1];          \
  static struct vtable *prevVT = 0;                        \
  static      method_t  method = 0;                        \
  (thisVT == prevVT                                        \
    ?  method                                              \
    : (prevVT = thisVT,                                    \
       method = bind(o, (MSG))))(o, ##ARGS);               \
})
```

Fig. 14. Optimising `send` with an inline cache. The `send` macro memoizes the previous vtable and associated closure returned from `bind`. `bind` is only called (and the memoized closure and vtable values updated) if the invocation is to an object whose vtable is not the same as the previous object's vtable at the same invocation site; otherwise the previously bound closure is reused immediately. This is safe provided the method name is a constant at any given invocation site.

```
struct entry {
  struct vtable  *vtable;
  struct object  *message;
  method_t        method;
} MethodCache[8192];

struct method_t *bind(struct object *obj,
                      struct object *msg)
{
  method_t        m;
  struct vtable  *vt     = obj->_vt[-1];
  unsigned long   offset = hash(vt, msg) & 8191;
  struct entry   *line   = MethodCache + offset;
  if (line->vtable == vt && line->message == msg)
    return line->method;
  m = ((msg == s_lookup) && (obj == vtable_vt))
    ? vtable_lookup(vt, msg)
    : send(vt, s_lookup, msg);
  line->vtable  = vt;
  line->message = msg;
  line->method  = m;
  return m;
}
```

Fig. 15. Optimising `bind` with a global method cache. The `MethodCache` stores vtables, message names, and the associated method implementations. To `bind` a message name within a vtable, a hash is computed from the vtable and name modulo the size of the method cache to create a cache line offset. If the vtable and name stored in the cache at that offset correspond to the vtable and name being bound, the stored method is returned immediately. Otherwise `lookup` is invoked in the vtable to bind the method name, and cache updated accordingly.

4 Extensions That Improve Generality

Section 3 described the simplest possible arrangement of the object model, in which each message name in a vtable is associated with the address of the native code of a corresponding method implementation. We found that the usefulness and generality of the object model were significantly improved by introducing an additional level of indirection, so that a message name is associated with a *closure*. Each closure contains two items: the address of the compiled code implementing the method and some (arbitrary) data, as shown in Figure 16.

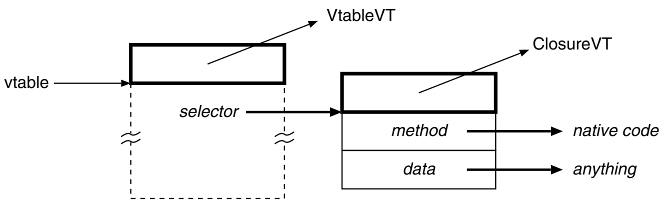


Fig. 16. Revised internals of vtables. A vtable maps message names onto closures, containing the address of the native code to be executed and some arbitrary data. Since closures are objects, they too have a pointer to a vtable describing their behaviour.

```
function vtable_addMethod(myClosure, self,
                                aSymbol, aMethod) =
  foreach i in 1 .. self.size
    if self.keys[i] = aSymbol
      self.values[i] := aMethod
      return
  self.keys.append(aSymbol)
  self.values.append(new closure(aMethod, nil))

function send(object, messageName, args...) =
  let closure = bind(object, messageName)
  return closure.method(closure, object, args...)

function bind(object, messageName) =
  let vt = object[-1]
  let closure =
    if messageName = lookup
          and object = VtableVT
      vtable_lookup(nil, vt, lookup)
    else
      send(vt, lookup, messageName)
  return closure
```

Fig. 17. Revised methods and functions. The method `addMethod` and the message sending functions `bind` and `send` are modified to store and retrieve closures instead of methods. Note that `addMethod`, like all method implementations, now accepts an additional argument (the closure in which it was found by `lookup`).

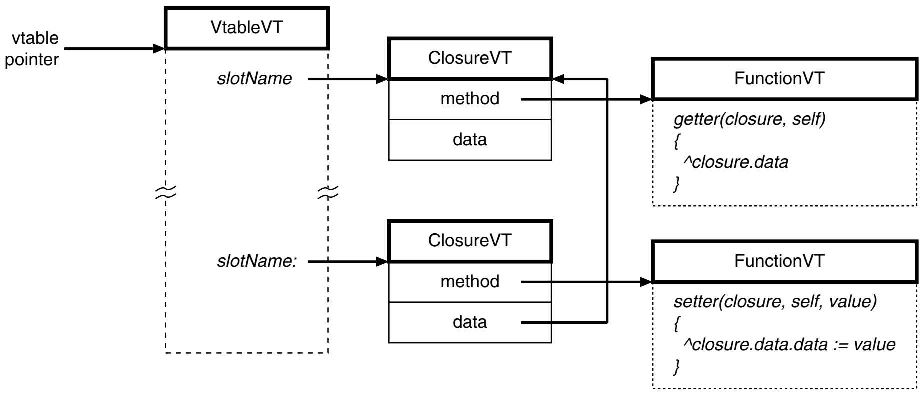


Fig. 18. Self-like slots. An assignable slot is implemented as a pair of methods: a 'getter' and a 'setter'. The value of the slot is stored as the `data` in the closure of its getter method. The `data` of the setter method's closure contains a reference to the getter's closure, allowing the setter to assign into the getter's `data`. A single implementation of getter and setter can be shared by all closures associated with assignable slots.

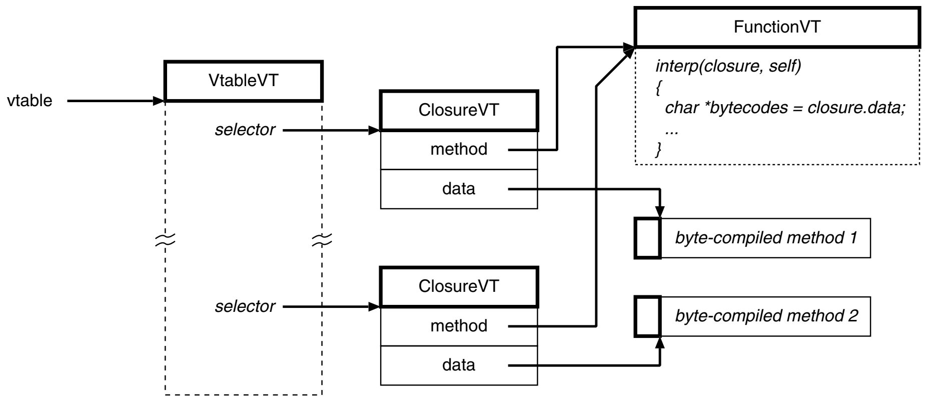


Fig. 19. Mixed-mode execution. An interpreter (for bytecodes or other structures) can be shared by any number of method closures. The structure to be interpreted is stored in the `data` part of the closure. As described in the text, the closure is passed as an argument to the method implementation (in this case the interpreter) from where its `data` is readily accessible. To the caller there is no difference between invoking a native method and invoking a byte-compiled method; the calling convention is the same.

The `bind` function is modified to return a closure as shown in Figure 17; `send` then invokes the method stored in the closure and passes the closure itself as an argument to the method (in addition to the message receiver and arguments). Method implementations are modified correspondingly, to accept the additional argument.

We believe the slight increase in complexity is more than justified by the generality that is gained. For example:

– Figure 18 shows how closures can be used as assignable slots, creating an end-user object model similar to that of traditional prototype-based languages.
– Figure 19 shows how closures are used to support mixed-mode execution [10]. A single interpreter method is shared between many closures whose `data` fields contain the code to be interpreted. To the caller there is no difference between invoking a natively compiled method and invoking an interpreted method.

Other useful extensions that have been implemented include support for 'Lieberman-style' prototypes [7] which provide much stronger encapsulation than the more common class-based inheritance. A detailed description of this extension is available online [9].

5 Evaluation

We validate the object model in two ways:

– By showing that it can be extended easily to support object models for existing languages or significant and useful features drawn from them. We do this by extending the prototype-based language (that uses the object model directly, as described in Section 2.2) first to support the Javascript object model (Section 5.1) and then by adding Traits (Section 5.2).
– By showing that its performance is sufficient for its use in serious language implementations (Section 5.3).

5.1 Ease of Use: Javascript Objects

Javascript [3] has a simple object model based on delegation [7] in which objects are dictionaries that map property names to their values. When an object is asked for an unknown property, it forwards the request to its prototype (fetched from its `__proto__` property). Properties are 'copy-on-write'; assigning to a property of an object either updates an existing property or creates a new property in the object. All objects, functions and methods in Javascript are based on this model.

Figure 20 shows one way of extending the object model to support these semantics. Note that this implementation is not intended to be used directly by programmers (although nothing prohibits this). Rather, a compiler is expected to translate Javascript expressions into method invocations. For example, a Javascript field access '`x.p`' is translated to '`x p`' (send message `p` to `x`, invoking its property getter). Similarly, the Javascript assignment '`x.p = y`' is translated to '`x set: #p to: y`' (send message `set:to:` to `x`, invoking its property setter) with arguments `#p` (the property name) and `y` (the new value).

```
vtable get [ ↑closure data ]

get := [ (vtable vtable lookup: #get) method ]

Object set: prop to: val
[
  | closure |
  (closure := self vtable lookup: prop) notNil
    ifFalse:
      [closure := self vtable methodAt: prop put: get].
  closure setData: val.
  prop == #__proto__
    ifTrue:
      [self vtable parent: val vtable.
       vtable flush].
]
```

Fig. 20. Javascript objects. Properties are implemented in a manner similar to that of slots in Figure 18. However, setter methods were eliminated in favour of a `set:to:` method that treats the `__proto__` property specially. If `__proto__` is assigned then the `parent` of the object's vtable is set to the `value`'s vtable, and any method caches are flushed. (Note that the block expression assigned to `get` is evaluated; the value assigned is the result of executing the block, not an unevaluated, literal block. Appendix A explains this syntax further.)

5.2 Ease of Use: Traits

Traits [11] are a powerful software composition mechanism. A trait is a collection of methods without state that can be manipulated and combined with other traits according to an algebra of composition, aliasing and exclusion. They are interesting because they provide the power of multiple inheritance without the complexity.

Figure 21 shows how the prototype-based language can be extended to support Traits. We can then easily implement the operations of the Traits 'algebra', for example:

```
Trait + aTrait
[
  ↑Trait delegated
    useTrait: self;
    useTrait: aTrait
]
```

This creates a new empty trait and adds both the receiver and the argument to it, composing their behaviours. (Method exclusion and method aliasing are left as an exercise; they take no more than a few minutes each. Once all three operations are available, you will have conforming traits implementation!)

```
Trait : Object ()

Object useTrait: aTrait [ aTrait addTo: self ]

Trait addTo: anObject [
  self vtable keysAndValuesDo: [:selector :closure |
    | newClosure |
    newClosure := anObject vtable
                    traitMethodAt: selector
                    put:           closure method.
    newClosure setData: closure data]
]

vtable traitMethodAt: aSelector put: aMethod [
  (self includesKey: aSelector)
    ifTrue: [↑self errorConflict: aSelector]
  ↑self methodAt: aSelector put: aMethod
]
```

Fig. 21. Support for traits. `Trait.addTo:` adds the methods of the receiver to the vtable of the argument. `vtable.traitMethodAt:put:` adds a method implementation with a given name to the receiver, and signals an error if the method name is already defined.

With the above traits implementation in place, we can write code such as:

```
T1 : Trait ()
T1 m  [ 'this is m' putln ]
T2 : Trait ()
T2 n  [ 'this is n' putln ]

C : Object ()  [ C useTrait: T1 + T2 ]

C o   [ self m; n ]
```

(Note that in the above what looks like a literal block after the declaration of `C` is actually an imperative; the program is executed from top to bottom, sending `useTrait:` to `C` before continuing with the installation of method `o` in `C`. Appendix A explains this further.)

5.3 Benchmarks

We measured the size and speed of a sample implementation written in GNU C (see Appendix B), faithfully following the algorithms and structure presented in this paper. All measurements were made on a 2.16 GHz Intel Core Duo.

The sample implementation is approximately 140 lines of code, containing:

- The three essential object types;
- One constructor function, for `symbols`;

– The five essential methods;
– Macros for `send` and `bind`, as presented in Section 3.2, with optional inline and global method caches; and
– An initialisation function that creates the initial objects and populates their vtables to create the object system as shown in Figure 13.

The object code size for all essential objects and their methods, with unoptimised `send` and `bind`, is 1,453 bytes. With the inline and global caches enabled, the code size grows to 1,822 bytes.[6] This should not be an issue for any but the most severely resource-constrained environments.

Next we investigate the overhead of dynamic dispatch through the vtables. We implemented the `nfibs` function (which has a very high ratio of message sends, or function invocations, to computation) in optimised C with statically-bound function calls and compared it with the object model using dynamically-bound message sends and an inline cache. The results from running `nfibs(34)` (performing 18,454,929 calls or method invocations) were:

type	*time*	*% of static call*
static call (C)	150 ms	100.0%
dynamic send	270 ms	55.6%

While the results are polluted a little by the arithmetic computation, they show that a static C function call is only approximately twice as fast as a dynamically-bound send through an inline cache. The actual overhead should be lower in practice since most code will perform more computation per call/send than `nfibs`.

Lastly, we implemented the example presented in Section 2 of this paper: data structures suitable for a Lisp-like language. We implemented a ‘traditional’ `length` primitive using a `switch` on an integer `tag` to select the appropriate implementation amongst a set of possible `case` labels. This was compared with an implementation in which data was stored using the object model and the `length` primitive used `send` to invoke a `method` in the objects themselves.[7] Both were run for one million iterations on forty objects, ten each of the four types that support the `length` operation. The results, with varying degrees of object model optimisations enabled, were:

implementation	*time*	*% of* `switch`
`switch`-based	503 ms	100.0%
dynamic object-based	722 ms	69.7%
+ global cache	557 ms	90.3%
+ inline cache	243 ms	207.0%

This shows that an extensible, object-based implementation can perform at better than half the speed of a typical C implementation for a simple language

[6] Darwin 8.8.1, Intel Core Duo, gcc-4.0.1 (Apple build 5367).

[7] A reference implementation, including the `length` benchmarks, can be downloaded from: `http://piumarta.com/software/id-objmodel`

primitive. With a global method cache (constant overhead, no matter how many method invocation sites exist) the performance is within 10% of optimised C. When the inline cache was enabled the performance was better than twice that of optimised C. In a practical language implementation the above performance gaps would be decrease in all cases as the amount of useful work per primitive increases. (It is hard to conceive of a simpler primitive than `length`.)

5.4 Limitations

The object model relies on a method cache [2] for performance. It is necessary to flush the cache after certain programming changes such as modifying a vtable (adding or removing a mapping, or storing into the `parent` slot). This is easy to do for both inline and global method caches, but is neither described in this paper nor counted in our evaluation of the sample implementation.

We do not count constructors in the number of methods in the object implementation. (There is no *requirement* for the constructors to be installed as methods although in practice it is convenient to do so.)

We also do not count the vtable pointer as part of the end-user object structure, since it appears before the nominal start of the object.

Lastly, the implementation of `bind` and `send` cannot be exposed as easily as the method lookup mechanism. This can be addressed by exposing the semantics of functions in the same way that the object model exposes the semantics of messaging (see Section 7). This permits almost unlimited flexibility to implement mechanisms such as multimethods.

6 Related Work

TinyObjects [6] also lets programmers remove limitations from the system instead of ‘programming around’ them. It provides a Metaobject Protocol (MOP) [5], at the end-user level of abstraction, that reflects on the implementation level and allows programmers to customise the object model to fit the needs of their applications. We address the same problem by implementing the object model and the equivalent of a very small MOP within a single level of abstraction. This way the programmer can directly manipulate the objects and methods that implement the semantics of their object model.

Smalltalk-80 [4] has methods (in classes Behaviour, Class and Metaclass) that provide what is essentially an incomplete MOP. While these can be used by programs (including the Smalltalk programming environment itself) to create new subclasses and modify method dictionaries, they cannot be used to modify the semantics of message sending itself nor the internal layout of objects.

McCarthy’s metacircular evaluator for LISP [8] demonstrated that it is possible for a language to be implemented (described) in itself. Such implementations are ‘open’: they allow programmers both to write ‘user programs’ and also to modify or extend the semantics of the language. The circular implementation of the object model brings an equivalent openness to the object-messaging paradigm.

Some systems, such as the Self programming language [12] and Lieberman's prototypes [7], present the user with simpler object models than the one we describe. The cost of this simplicity is that some of the semantics of their object models is hidden (slot lookup in particular) and cannot be modified by end user code. Self also requires a significantly more complex runtime to run efficiently [1]. The model is much closer Self's internal object model which uses *maps* (similar to vtables) to describe the behaviour of entire *clone families*. Very promising recent experiments with Self aim to expose the entire implementation to the programmer [13].

7 Conclusions and Further Work

We presented a simple, extensible object model that exposes its own semantics in terms of the objects and messages that it implements. This circularity in the implementation results in surprising flexibility; end users have direct access to, and control over, the implementation mechanisms of the object model itself. Our experience with this object model has shown that it can be extended easily to support powerful features such as sideways composition and mixed-mode execution. While it is not necessarily a friendly model for hand-written code, it is an attractive target for automatic translation. It could also be an attractive target for statically-typed languages, where the compiler can guarantee runtime type safety.

Because it imposes no structure on end user objects, the model invites experimentation that might otherwise be difficult. For example, it allows a pointer to a compiled native function to also be an object, to which messages can be sent; a vtable in the word before the function prologue suffices. We envisage going further and storing useful information about compiled code (stack layout, signature information, pre- and post-conditions, etc.) in the word before the function's vtable pointer (at offset -2).

This complements ongoing work with dynamic code generation that brings the functional aspects of the object model (method implementations, method invocation, and `send` and `bind` in particular) under the control of the programmer. This work will be the subject of forthcoming publications.

Starting with the algorithms and C language bindings described in this paper, implementing the object model in C took approximately three hours. The essential objects and methods total 140 lines of source code. Not only is it tiny, but it also scales well: in a slightly different form it has been in daily use by several people for over a year. This model provides rich Smalltalk-like class libraries, implements its own compiler and dynamic code generator for multiple architectures, and integrates seamlessly with platform libraries and garbage collection. With the addition of a few lines of code it can support tagged immediate quantities, and represent the object `nil` with the NULL pointer.

References

1. Chambers, C., Ungar, D., Lee, E.: An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In: OOPSLA 1989 Conference proceedings on Object-oriented programming systems, languages and applications, pp. 49–70. ACM Press, New York (1989)
2. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the Smalltalk-80 system. In: POPL 1984: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 297–302. ACM Press, New York (1984)
3. ECMA. Ecmascript language specification (December 1999), `http://www.ecma.ch/ecma1/stand/ecma-262.htm`
4. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., Boston (1983)
5. Kiczales, G., des Rivieres, J., Bobrow, D.G.: The art of metaobject protocol. MIT Press, Cambridge (1991)
6. Kiczales, G., Paepcke, A.: Open Implementations and Metaobject Protocols, `http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-TUT95/for-web.pdf`
7. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: OOPLSA 1986: Conference proceedings on Object-oriented programming systems, languages and applications, pp. 214–223. ACM Press, New York (1986)
8. McCarthy, J.: LISP 1.5 Programmer's Manual. The MIT Press, Cambridge (1962)
9. Piumarta, I.: Efficient Sideways Composition via 'Lieberman' Prototypes. VPRI Research Memo RM-2007-002-a (2007), `http://vpri.org/pdf/lieberman_proto_RM-2007-002-a.pdf`
10. Sankar, S., Viswanadha, S., Solorzano, J.H., Duncan, R.J., Bacon, D.J.: Mixed-mode execution for object-oriented programming languages. US Patent 6854113, issued February 8 (2005), `http://www.patentstorm.us/patents/6854113.html`
11. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 248–274. Springer, Heidelberg (2003)
12. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: OOPSLA 1987: Conference proceedings on Object-oriented programming systems, languages and applications, pp. 227–242. ACM Press, New York (1987)
13. Ungar, D., Spitz, A., Ausch, A.: Constructing a metacircular virtual machine in an exploratory programming environment. In: OOPSLA 2005: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 11–20. ACM Press, New York (2005)

A Prototype Language Syntax

The prototype-based language used for several examples in the text has a syntax similar to that of Smalltalk-80 [4] with a few significant differences described here.

A.1 Type Declarations

New types are introduced by creating a named prototype of that type. For example,

```
Derived : Base ( a b )
```

creates a variable '`Derived`' (in a kind of 'global namespace') and assigns to it a new prototype belonging to a family of objects that inherit behaviour and state from the family of '`Base`' (another prototype) and which extend that state with two new slots called `a` and `b`. The new vtable for `Derived` is created automatically by sending `delegated` to the vtable for `Base`; this vtable is then sent the message `allocate` to create the prototype stored in `Derived`.

A.2 Method Definitions

The body of a method follows its defining message pattern within square brackets. For example,

```
Derived frobble: bob with: bill
[
    ↑bob frobbleWith: bill from: self
]
```

installs the method `frobble:with:` in the vtable for `Derived` by sending it the message `addMethod` with the message name and method implementation as arguments.

A.3 Top-Level Statements

Arbitrary statements can be executed at the 'top-level' of the program (anywhere a definition is allowed) by enclosing them in square brackets. For example,

```
[
    'running DeepThought program...' putln.
    DeepThought new multiply: 6 by: 9.
]
```

announces to the user that an application is about to run, then instantiates and runs it.

A.4 Top-Level Definitions

Variables in the 'global namespace' can be bound to arbitrary values (not just to new prototypes as described above). For example,

```
TheAnswer := [ 42 ]
```

creates a 'global' variable named `TheAnswer` and initialises it with the value of the last expression in the block (in this case, the literal `42`).

B Sample Object Model Implementation

```
/*  A sample implementation in GNU C of the object model described
 *  in this paper.  This code, and that of the benchmarks discussed
 *  in the text, can be downloaded from:
 *    http://piumarta.com/software/id-objmodel
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ICACHE 1   /* nonzero to enable point-of-send inline cache */
#define MCACHE 1   /* nonzero to enable global method cache        */

struct vtable;
struct object;
struct symbol;

typedef struct object *oop;
typedef oop *(*method_t)(oop receiver, ...);

struct vtable
{
  struct vtable  *_vt[0];
  int             size;
  int             tally;
  oop            *keys;
  oop            *values;
  struct vtable  *parent;
};

struct object {
  struct vtable *_vt[0];
};

struct symbol
{
  struct vtable *_vt[0];
  char          *string;
};

struct vtable *vtable_vt   = 0;
struct vtable *object_vt   = 0;
struct vtable *symbol_vt   = 0;

oop s_addMethod = 0;
oop s_allocate  = 0;
oop s_delegated = 0;
oop s_lookup    = 0;
oop s_intern    = 0;

oop symbol      = 0;

struct vtable *SymbolList  = 0;

extern inline void *alloc(size_t size)
{
RR  struct vtable **ppvt=
    (struct vtable **)calloc(1, sizeof(struct vtable *) + size);
  return (void *)(ppvt + 1);
}

oop symbol_new(char *string)
{
  struct symbol *symbol = (struct symbol *)alloc(sizeof(struct symbol));
  symbol->_vt[-1] = symbol_vt;
  symbol->string = strdup(string);
  return (oop)symbol;
}

oop vtable_lookup(struct vtable *self, oop key);

#if MCACHE
struct entry {
  struct vtable  *vtable;
  struct object  *selector;
  method_t        method;
} MethodCache[8192];
#endif

#if ICACHE
# define send(RCV, MSG, ARGS...) ({                                 \
             oop            r      = (struct object *)(RCV); \
             struct vtable *thisVT = r->_vt[-1];             \
      static struct vtable *prevVT = 0;                      \
      static      method_t  method = 0;                      \
      (thisVT == prevVT                                      \
        ? method                                             \
        : (prevVT = thisVT,                                  \
           method = _bind(r, (MSG))))(r, ##ARGS);            \
    })

#else /* !ICACHE */
# define send(RCV, MSG, ARGS...) ({       \
      oop      r      = (oop)(RCV);       \
      method_t method = _bind(r, (MSG));  \
      method(r, ##ARGS);                  \
    })
#endif

method_t _bind(oop rcv, oop msg)
{
  method_t       method;
  struct vtable *vt = rcv->_vt[-1];
#if MCACHE
  unsigned int   hash =
    (((unsigned)vt << 2) ^ ((unsigned)msg >> 3))
    & ((sizeof(MethodCache) / sizeof(struct entry)) - 1);
  struct entry  *line = MethodCache + hash;
  if (line->vtable == vt && line->selector == msg)
    return line->method;
#endif
  method = ((msg == s_lookup) && (rcv == (oop)vtable_vt))
    ? (method_t)vtable_lookup(vt, msg)
    : (method_t)send(vt, s_lookup, msg);
#if MCACHE
  line->vtable   = vt;
  line->selector = msg;
  line->method   = method;
#endif
  return method;
}

oop vtable_allocate(struct vtable *self, int payloadSize)
{
  struct object *object = (oop)alloc(payloadSize);
  object->_vt[-1] = self;
  return object;
}

struct vtable *vtable_delegated(struct vtable *self)
{
  struct vtable *child =
    (struct vtable *)vtable_allocate(self, sizeof(struct vtable));
  child->_vt[-1] = self ? self->_vt[-1] : 0;
  child->size    = 2;
  child->tally   = 0;
  child->keys    = (oop *)calloc(child->size, sizeof(oop));
  child->values  = (oop *)calloc(child->size, sizeof(oop));
  child->parent  = self;
  return child;
}
```

```
oop vtable_addMethod(struct vtable *self, oop key, oop method)
{
  int i;
  for (i = 0;  i < self->tally;  ++i)
    if (key == self->keys[i])
      return self->values[i] = (oop)method;
  if (self->tally == self->size)
    {
      int sz= (self->size *= 2);
      self->keys   = (oop *)realloc(self->keys,   sizeof(oop) * sz);
      self->values = (oop *)realloc(self->values, sizeof(oop) * sz);
    }
  self->keys  [self->tally  ] = key;
  self->values[self->tally++] = method;
  return method;
}

oop vtable_lookup(struct vtable *self, oop key)
{
  int i;
  for (i = 0;  i < self->tally;  ++i)
    if (key == self->keys[i])
      return self->values[i];
  if (self->parent)
    return send(self->parent, s_lookup, key);
  fprintf(stderr, "lookup failed %p %s\n",
          self, ((struct symbol *)key)->string);
  return 0;
}

oop symbol_intern(oop self, char *string)
{
  oop symbol;
  int i;
  for (i = 0;  i < SymbolList->tally;  ++i)
    {
      symbol = SymbolList->keys[i];
      if (!strcmp(string, ((struct symbol *)symbol)->string))
        return symbol;
    }
  symbol = symbol_new(string);
  vtable_addMethod(SymbolList, symbol, 0);
  return symbol;
}

void init(void)
{
  vtable_vt = vtable_delegated(0);
  vtable_vt->_vt[-1] = vtable_vt;
  object_vt = vtable_delegated(0);
  object_vt->_vt[-1] = vtable_vt;
  vtable_vt->parent = object_vt;

  symbol_vt = vtable_delegated(object_vt);
  SymbolList = vtable_delegated(0);

  s_lookup = symbol_intern(0, "lookup");
  vtable_addMethod(vtable_vt, s_lookup,    (oop)vtable_lookup);

  s_addMethod = symbol_intern(0, "addMethod");
  vtable_addMethod(vtable_vt, s_addMethod, (oop)vtable_addMethod);

  s_allocate = symbol_intern(0, "allocate");
  send(vtable_vt, s_addMethod, s_allocate,  vtable_allocate);
  symbol = send(symbol_vt, s_allocate, sizeof(struct symbol));

  s_intern = symbol_intern(0, "intern");
  send(symbol_vt, s_addMethod, s_intern, symbol_intern);

  s_delegated = send(symbol, s_intern, (oop)"delegated");
  send(vtable_vt, s_addMethod, s_delegated, vtable_delegated);
}
```

The Lively Kernel A Self-supporting System on a Web Page

Daniel Ingalls[1], Krzysztof Palacz[1], Stephen Uhler[1], Antero Taivalsaari[2], and Tommi Mikkonen[2]

[1] Sun Microsystems Laboratories, Menlo Park, CA
[2] Sun Microsystems Laboratories, Tampere, Finland
Lively@Sun.com

Abstract. The Lively Kernel is a complete platform for Web programming written in JavaScript™ using graphics available in leading browsers. A widget set built from these elements provides a user interface kit, and the widget set is also extensible. A window-based IDE allows users to edit their applications and even the system itself. When a user visits the Lively Kernel page,

`http://research.sun.com/projects/lively/index.xhtml`

the kernel loads and runs with no installation whatsoever. The user can immediately construct new objects or applications and manipulate the environment.

The Lively Kernel is able to save its creations, and even clone itself, onto Web pages. In so doing, it defines a new form of dynamic content on the Web. Moreover, since it can run in today's browsers, it promises that wherever there is the Internet, there can be authoring of Web content.

Beyond its utility, the simplicity and completeness of the Lively Kernel make it a practical benchmark of system complexity, and a flexible laboratory for exploring new approaches to security, simplified graphics, and Web technologies in general.

Keywords: Dynamic language, JavaScript, Morphic, self-supporting, Web programming, rich internet applications, widgets, Web 2.0.

Note to Readers: As of this writing, the Lively Kernel runs with no installation in the Firefox 3 beta and Safari 3 browsers. We are preparing an applet that will allow it to run in other browsers until their internal graphics are adequate for install-free operation.

1 Time for a Change

There is no good reason for Web Programming to be more complicated, less general, or any less fun than other modes of programming. There are reasons,

JavaScript is a registered trademark of Sun Microsystems, Inc.

of course, mainly focusing on static content and markup languages and ignoring several decades of experience with lean computing kernels built around Lisp, Smalltalk, and other dynamic languages. That is history, but it need not hold us back. In this paper we describe a simple and general kernel for programming the Web. Its core is less than 10,000 lines of code (with comments), it runs in major browsers with no installation, and it performs well. We call it the Lively Kernel.

In this paper, we begin by the observation that, completely apart form the text-based world of HTML and its decorations, the now ubiquitous Internet browsers provide all that is needed for a rebirth of active objects in the Web context. Beginning with the JavaScript language and standard browser graphics, we trace the construction of a computing environment from basic shapes to widgets (active user interface components) to programming tools, ending with an environment is self-supporting and that supports general application development and deployment on the Internet.

Look at a typical Web page on a typical computer and you will see static graphics, most likely generated from a decades-old markup language, being presented by a computer capable of executing a billion instructions per second. There is something wrong with that picture. There is no reason that the entire page cannot be an active object, ready to respond in all the general ways that computers were built to support. This is the Lively Kernel view of the Web and Web programming. Ironically it is not even a new approach, but rather the tried and true approach of numerous dynamic programming environments that were in widespread use long before HTML was adopted as the standard of Web content presentation.

We observe that every browser supports a dynamic programming language, one or more graphics systems, and support for network communication. While JavaScript has been mainly shaped by its role as a scripting vehicle for HTML, it is actually a perfectly usable dynamic programming language. In the area of graphics, most browsers support HTML, a flat graphics model (Canvas) and a retained graphics model (SVG; see `http://www.w3.org/TR/SVG11/`). For communication, modern browsers offer XmlHttpRequest for access to remote hosts elsewhere on the Internet. To a self-supporting system builder, this is all one needs.

We inherit from the World Wide Web an architecture built around a text markup language. The Lively Kernel sets that architecture aside in favor of modern graphics and a dynamic programming language. We begin by turning the conventional Web programming "stack" upside down as shown in figure 1.

The first priority of this architecture is to provide a world of active objects. This is accomplished by putting a dynamic language close to the operating system, which allows both the infrastructure (widgets, etc.) and the application to share the same pervasive generality and power. The compactness and capability of our system validates this approach.

We began with a few simple experiments with shapes on a Web page made active in small ways by attached JavaScript methods. Encouraged by the responsiveness of both JavaScript and our graphics layer, we set about implementing a more complete framework for active graphical objects on a Web page. We chose

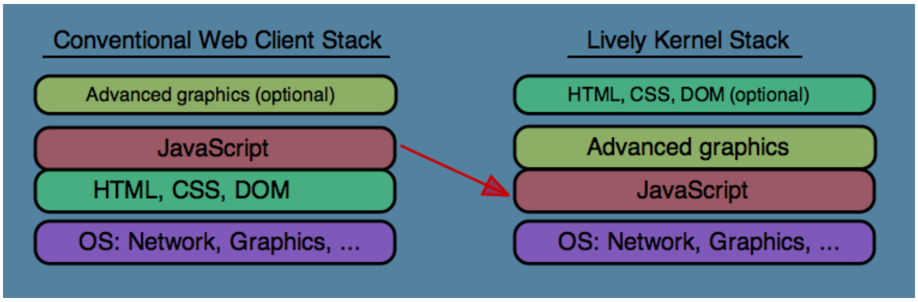

Conventional Web Client Stack
Advanced graphics (optional)
JavaScript
HTML, CSS, DOM
OS: Network, Graphics, ...
Lively Kernel Stack
HTML, CSS, DOM (optional)
Advanced graphics
JavaScript
OS: Network, Graphics, ...

Fig. 1. Turning Web programming upside down

to follow the Morphic architecture, which we knew from both the Self and Squeak programming environments, and which we consider to be a model of simplicity and generality.

2 A Quick Summary of the Morphic Architecture

The Morphic architecture is very simple. It defines a class of graphical objects, or "morphs", each of which has some or all of the following properties:

- A shape, or graphical appearance
- A set of submorphs, used to construct the "scene graph" of the page or world
- A coordinate transformation that affects its shape and any submorphs
- An event handler for mouse and keyboard events
- An editor for changing its shape
- A layout manager for laying out its submorphs
- A stepping protocol for time-varying behavior
- A damage region and repainting protocol and double-buffered display mechanism when this is not available in the underlying graphics

A few other high-level morphs serve to complete a meaningful graphical environment. WorldMorph captures the notion of an entire screen view (often a Web page); its shape defines its background appearance, and its submorphs comprise the remaining content of the page. A world has a scheduler for managing user input, external input, and timer-based events. A HandMorph is the Morphic manifestation of a cursor; it can be used to pick up, move, and deposit other morphs. Its shape may change to indicate different cursor states, and it is the source of user events.

A property, that can be enabled or not, causes dropping of one morph upon another to make the first a submorph of the second. Mashups and new widgets or complete user interfaces can be assembled in this concrete manner.

In the Lively Kernel, a Morphic world may have several hands active at the same time, corresponding to multiple collaborating users of that world, and multiple worlds may be linked in the manner of linked Web pages.

Interested readers are referred to the original papers on Morphic [12], and to the Lively Kernel technical documentation.

3 A Lively Construction

In contrast with the static elements of most Web pages, each element of a Lively Web page is a Morphic object able to be picked up, moved, duplicated and reshaped. Thus, at its simplest, the Lively Kernel functions as a rudimentary graphics editor. The sequence shown in figure 2 illustrates the construction of a simple truck shape by concrete manipulation. In figure 2a we see a palette of useful shapes. In figure 2b, the rectangle has been copied, and extended in figure 2c. In figure 2d, the rectangle has been colored yellow, an ellipse has been copied, resized, and colored, and has been given a thick black border to resemble a tire, and a second copy has already been affixed to the bus. In figure 2e, the truck is complete, and figure 2f shows a family of trucks, copied and rotated from the new master, all operations that can be accomplished with simple gestures in the Lively Kernel's graphic editor. The next section will show how similar structures are built programmatically.

Beyond the basic vocabulary of the underlying graphics support, the programmability of Morphic shapes provides an unlimited range of graphical idioms.

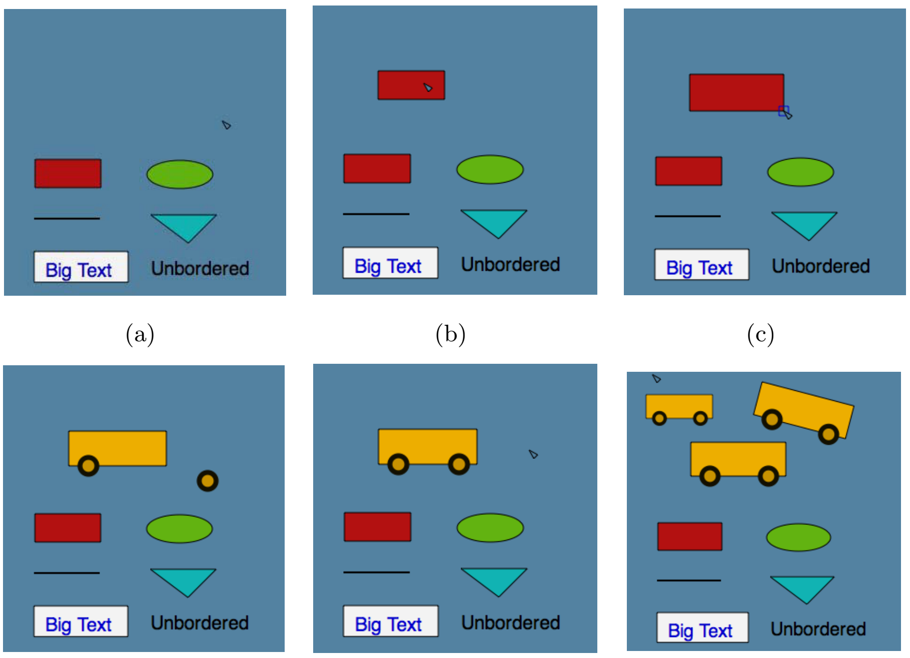
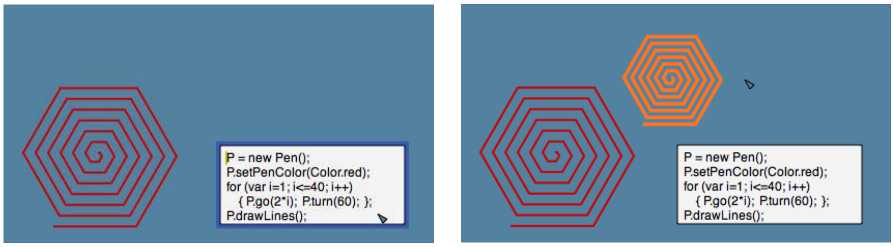


Fig. 2. Drag-and-drop construction of simple objects in the Lively Kernel



```
P = new Pen();
P.setPenColor(Color.red);
for (var i=1; i<=40; i++)
  { P.go(2*i); P.turn(60); };
P.drawLines();
```

```
P = new Pen();
P.setPenColor(Color.red);
for (var i=1; i<=40; i++)
  { P.go(2*i); P.turn(60); };
P.drawLines();
```

(a) (b)

Fig. 3. Extending the graphical vocabulary

Figure 3 shows a small snippet of JavaScript that draws a polygonal spiral. Beyond mere marks on the screen, this code produces a fully active object that can be copied, scaled, and colored, as shown, and that could be set to spinning with one more line of code, such as

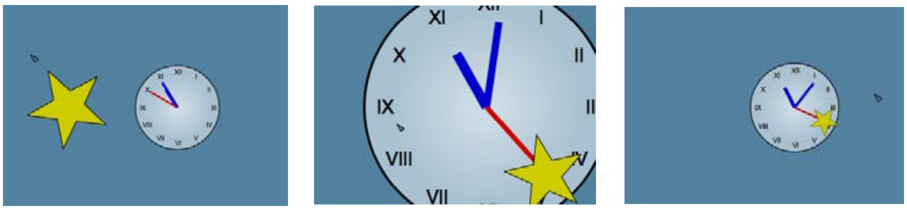
```
    this.startStepping(50, \rotateBy", 0.1); //0.1 radians every 50 ms
```

4 A Lively Clock

Here we present a simple application written in the Lively Kernel. The purpose is to illustrate the style of code written, and the advantages derived from the underlying architecture. Most of the code is in the "`makeNewFace`" method which centers the hour labels at equally spaced points around the face, and creates the three hands. Once the clock is created, the remaining task is to make the hands move. This is accomplished in the "`setHands`" method. Note that no code is required to update the image. It suffices to set the rotation of the hands appropriately; the architecture takes care of any required redrawing. The "`setHands`" method is scheduled to be called every 1000 milliseconds by the "`startSteppingScripts`" method, which is invoked whenever a new morph is placed into the world.

```
// ================================
// A Lively Kernel clock
// ================================
ClockMorph = Class.create(Morph, {
    defaultBorderWidth: 2,
    type: "ClockMorph",

initialize: function($super, position, radius) {
    $super(position.asRectangle().expandBy(radius), "ellipse");
    this.openForDragAndDrop = false;
    this.linkToStyles(['clock']);
    this.makeNewFace();
    return this;
},
makeNewFace: function() {
    var bnds = this.shape.bounds();
    var radius = bnds.width/2;
    var fontSize = Math.max(
        Math.floor(0.04 * (bnds.width + bnds.height)),2);
    var labelSize = fontSize; // room to center with default inset

    for (var i = 0; i < 12; i++) { // Place the 12 labels...
    var labelPosition = bnds.center().addPt(
        Point.polar(radius*0.85,((i-3)/12)*Math.PI*2)).addXY(labelSize,0);
    var label = new TextMorph(pt(0,0).extent(pt(labelSize*3,labelSize)),
    // (i>0 ? i : 12) + ""); // English numerals
    // Roman:
    ['XII','I','II','III','IV','V','VI','VII','VIII','IX','X','XI'][i]);
    label.setWrapStyle(WrapStyle.SHRINK);
    label.setFontSize(fontSize); label.setInset(pt(0,0));
    label.setBorderWidth(0); label.setFill(null);
    label.align(label.bounds().center(),labelPosition.addXY(-1,1));
    this.addMorph(label);
    }
    this.addMorph(this.hourHand =
        Morph.makeLine([pt(0,0),pt(0,-radius*0.5)],4,Color.blue));
    this.addMorph(this.minuteHand =
        Morph.makeLine([pt(0,0),pt(0,-radius*0.7)],3,Color.blue));
    this.addMorph(this.secondHand =
        Morph.makeLine([pt(0,0),pt(0,-radius*0.75)],2,Color.red));
    this.setHands();
    this.changed();
},

setHands: function() { // Set hand angles from time
    var now = new Date();
    var second = now.getSeconds();
    var minute = now.getMinutes() + second/60;
    var hour = now.getHours() + minute/60;
    this.hourHand.setRotation(hour/12*2*Math.PI);
    this.minuteHand.setRotation(minute/60*2*Math.PI);
    this.secondHand.setRotation(second/60*2*Math.PI);
},

startSteppingScripts: function() { // Called when placed in a world
    this.startStepping(1000, "setHands"); // once per second
}
});
```

Listing 1.1. A Morphic Clock in the Lively Kernel



(a) (b) (c)

Fig. 4. Composition of dynamic objects in the Lively Kernel

We have already pointed out the architectural advantage provided in redrawing of the hands and periodic execution of the scheduled behavior. In addition, the entire construct inherits the ability to be scaled and rotated arbitrarily.

Figures 4a to 4c illustrate the flexibility of the Lively application architecture. In figure 4a we see a clock next to a (spinning) star. In figure 4b, the clock has been expanded to a large size (note that the text looks better, not worse), making it possible to drop the spinning star onto the end of the second hand. In figure 4c, we see the clock shrunk back to its normal size, but still sporting a spinning star on the end of its second hand. Of course, it is always possible to disable this kind of fanciful manipulation, but at this point we are exploring flexibility, not trying to prevent it.

The clock is a graphical assembly of text, lines and an ellipse, together with a simple script that endows the assembly with real clock-ness. In an equally simple manner, a basic set of “widgets” (common active user interface components) can be built up from the basic shapes plus a few simple methods. If the earlier truck example illustrated the construction of new molecules, then this is a bit like chemistry since, besides the mere assembly of parts, there is a meaningful model under each widget, and the interaction of those underlying values is the beginning of open-ended computing and self-support.

5 Lively Development Tools

Within the Morphic context, we chose to implement the Lively Kernel’s widget set with a model/view separation along the lines of many Smalltalk systems. This choice was influenced by experience with GUI-builder applications and the Fabrik visual programming system. Besides allowing for multiple views of a given model, the model/view separation makes it easier to infer appropriate model structure from a given concrete assembly of UI components. It also turns out to provide a flexibility that is vital for migration of functionality between client and server where this is desired.

Figure 5 shows a number of simple widgets arranged in a test panel. It is not much to look at as a picture, but if one runs the Lively Kernel from our site, one finds that the buttons, text boxes and lists are in groups that exhibit bidirectional coupling through their shared models. For instance the slider is

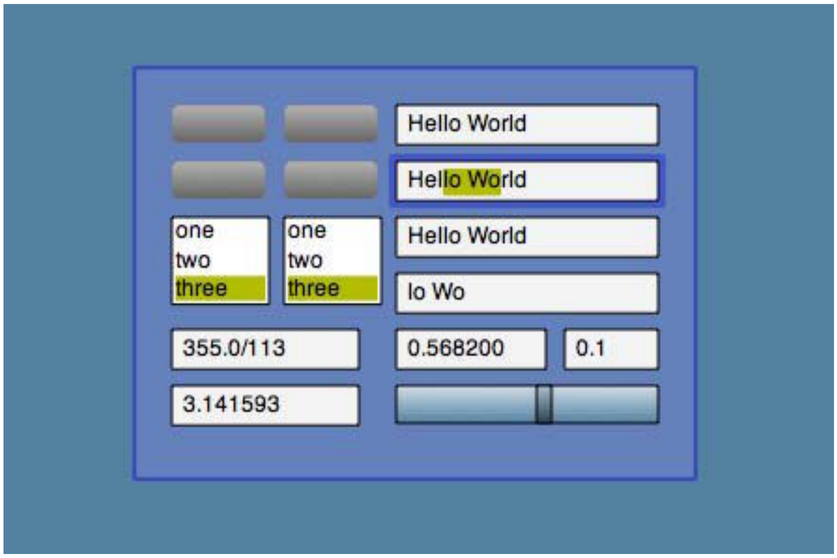
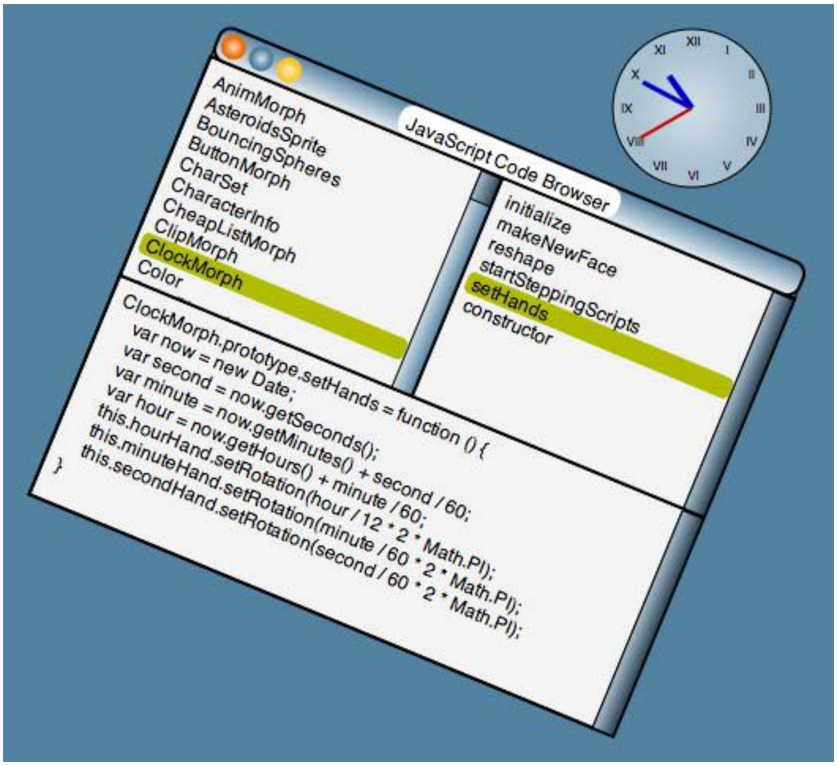


Fig. 5. A text panel for widgets with shared models



```
ClockMorph.prototype.setHands = function () {
  var now = new Date;
  var second = now.getSeconds();
  var minute = now.getMinutes() + second / 60;
  var hour = now.getHours() + minute / 60;
  this.hourHand.setRotation(hour / 12 * 2 * Math.PI);
  this.minuteHand.setRotation(minute / 60 * 2 * Math.PI);
  this.secondHand.setRotation(second / 60 * 2 * Math.PI);
}
```

Fig. 6. A Code Browser viewing the Clock application

hooked to a numerical model that is bidirectionally connected to one of the text views with a read/print converter.

We made the claim above that even a rudimentary assembly of widgets is the makings of open-ended computing. This should not surprise any of our readers, and we see in Figure 6 a piece of almost professional-looking software which is little more than an assembly of text boxes, lists, a clipping component, and the same slider shown in Figure 5, now doing service in (almost) vertical orientation as a scroll bar.

If we look more closely at the list on the left – "`ButtonMorph`", "`ClipMorph`" – these are names of classes in the system itself, as are the selections, "`ClockMorph`", and "`setHands`". It is in fact a code browser (as its title bar confirms) written in the system, and viewing code in the system; in fact the very code exhibited earlier for the clock.

Owing to the uniform graphics architecture, the browser application too can be used at any scale or rotation. If you are running the Lively Kernel, you may find this browser in the "Development Tools" world. Browse to this same method, add a minus sign to the parameter of the last setRotation call, and you will have a clock whose second hand runs backwards.

The code browser shown above is scarcely larger than than the ClockMorph class it is editing. We exhibit it here, if only to show that, having built up a rudimentary set of widgets, self-support with a graphical user interface is not so difficult to achieve.

```
// ================================
// A Lively Kernel code browser
// ================================
Widget.subclass('SimpleBrowser', {
  defaultViewTitle: "Javascript Code Browser",
  pins: ["+ClassList", "-ClassName", "+MethodList",
      "-MethodName", "MethodString"],

  initialize: function($super) {
     var model = new SyntheticModel(this.pins);
     var plug = model.makePlugSpecFromPins(this.pins);
     $super(plug);
     this.scopeSearchPath = [Global];
     model.setClassList(this.listClasses());
  },

  updateView: function(aspect, source) {
    var p = this.modelPlug;
    if (!p) return;
    switch (aspect) {
    case p.getClassName:
      var className = this.getModelValue('getClassName');
      this.setModelValue("setMethodList",this.listMethodsFor(className));
      break;
     case p.getMethodName:
        var methodName = this.getModelValue("getMethodName");
        var className = this.getModelValue("getClassName");
        this.setModelValue("setMethodString",
            this.getMethodStringFor(className, methodName));
     break;
     case p.getMethodString:
     this.getModelValue("getMethodString"));
     break;
     }
 },

  listClasses: function() {
    var list = [];
    for (var i = 0; i < this.scopeSearchPath.length; i++) {
        var p = this.scopeSearchPath[i];
        var scopeCls = [];
        Class.withAllClassNames(p, function(name) {scopeCls.push(name);});
        list = list.concat(scopeCls.sort());
    }
    return list;
},

    listMethodsFor: function(className) {
        if (className == null) return [];
        var sorted = (className == 'Global')
            ? Global.constructor.functionNames().without(className).sort()
            : Global[className].localFunctionNames().sort();
        var defStr = "*definition";
        var defRef = SourceControl &&
            SourceControl.getSourceInClassForMethod(className, defStr);
        return defRef ? [defStr].concat(sorted) : sorted;
    },

    getMethodStringFor: function(className, methodName) {
        if (!className || !methodName) return "no code";
        else return Function.methodString(className, methodName);
    },

setMethodString: function(newDef) { eval(newDef); },

buildView: function(extent) {
    var panel = PanelMorph.makePanedPanel(extent, [
        ['leftPane', newTextListPane, new Rectangle(0, 0, 0.5, 0.5)],
        ['rightPane', newTextListPane, new Rectangle(0.5, 0, 0.5, 0.5)],
        ['bottomPane', newTextPane, new Rectangle(0, 0.5, 1, 0.5)]
    ]);
    var model = this.getModel();
     panel.leftPane.connectModel( {model: model, getList: "getClassList",
         setSelection: "setClassName"});
     panel.leftPane.updateView("getClassList");
     panel.rightPane.connectModel({model: model, getList: "getMethodList",
         setSelection: "setMethodName"});
     panel.bottomPane.connectModel({model: model,
         getText: "getMethodString", setText: "setMethodString"});
     return panel;
 }
});
```

Listing 1.2. A simple code browser

Most of the browser code should be fairly self-explanatory. We point out that the reference to SourceControl allows this same browser to function stand-alone, reflecting on the sources of the running system (JavaScript functions will print themselves in response to `toString()`) or, in a team programming environment, it will access the original source code files in a repository. The `connectModel()` protocol in the `buildView` method provides for "pluggable" views so that, for example, the two list panes are connected to different aspects of the underlying model.

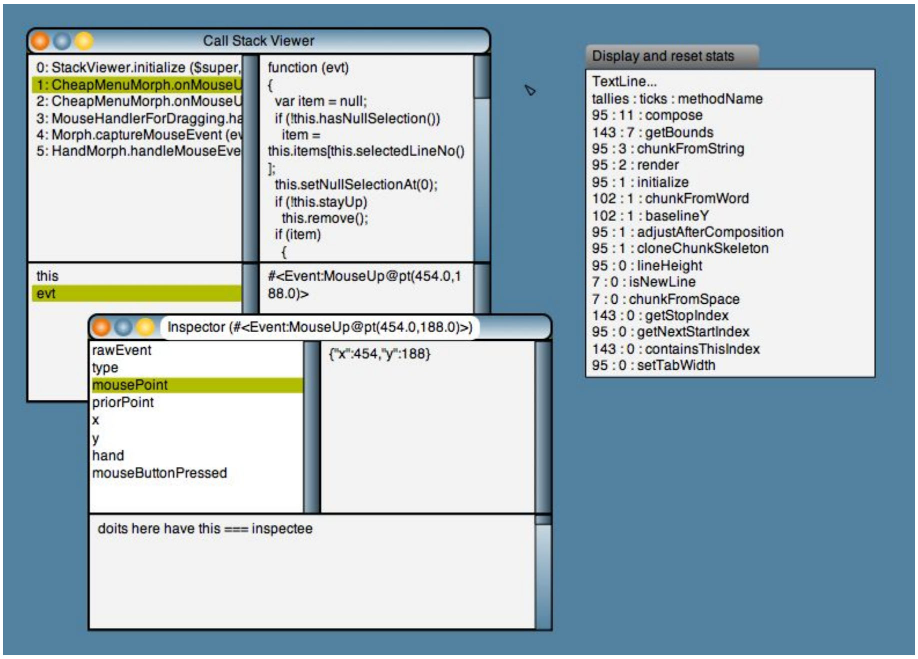
6 More Tools for Self-support

While a source code browser is the hallmark of self-support in any system, a number of other reflective tools are useful in the maintenance and evolution of a software system. The Lively Kernel provides a number of these, including, an Object Inspector, a Stack Viewer, and a Profiler. Examples of these tools appear in figure 7.

The Profiler and Stack Viewer are perhaps the most interesting of our reflective tools, because the normal JavaScript environment is missing the necessary reflection to provide them. However, the resourceful software engineer will find just enough reflection to provide these tools.

Consider the problem of execution time analysis: we wish to know exactly how many times each method is invoked in the course of some computation, either for rigor, or to understand where most of the time is spent. In the latter case, we would ideally like to see an accounting of the real time spent in each method as well. JavaScript engines provide neither of these reports, but they do, at least, give us a millisecond clock.

Listing 1.3 shows how the millisecond time can be used to provide a relatively complete profiler in a dynamic language environment. The essence of this function is simply an enumeration of all the methods in a class. If invoked with the parameter "`start`", then it replaces every method with an anonymous wrapping function (`tallyFunc`) that, after some bookkeeping, calls the original method, and if called with "`stop`", it undoes this replacement. The bookkeeping, in this case, involves incrementing a tally count by one, and a ticks count by the number of millisecond ticks between call and return of the method. The remaining parameters for the outer call use the same enumeration to reset the tallies, or to collect them for reporting, as in figure 7. The button above the profile is a



```
0: StackViewer.initialize ($super,
1: CheapMenuMorph.onMouseU
2: CheapMenuMorph.onMouseU
3: MouseHandlerForDragging.ha
4: Morph.captureMouseEvent (ev
5: HandMorph.handleMouseEve

this
evt
```

```
function (evt)
{
 var item = null;
 if (!this.hasNullSelection())
  item =
this.items[this.selectedLineNo()
];
 this.setNullSelectionAt(0);
 if (!this.stayUp)
  this.remove();
 if (item)
  {

#<Event:MouseUp@pt(454.0,1
88.0)>
```

```
rawEvent
type
mousePoint
priorPoint
x
y
hand
mouseButtonPressed
```

```
{"x":454,"y":188}
```

```
doits here have this === inspectee
```

```
TextLine...
tallies : ticks : methodName
95 : 11 : compose
143 : 7 : getBounds
95 : 3 : chunkFromString
95 : 2 : render
95 : 1 : initialize
102 : 1 : chunkFromWord
102 : 1 : baselineY
95 : 1 : adjustAfterComposition
95 : 1 : cloneChunkSkeleton
95 : 0 : lineHeight
7 : 0 : isNewLine
7 : 0 : chunkFromSpace
143 : 0 : getStopIndex
95 : 0 : getNextStartIndex
143 : 0 : containsThisIndex
95 : 0 : setTabWidth
```

Fig. 7. An Object Inspector, Stack Viewer, and Profiler

very simple control: each time it is pressed, it reads out the tallies and the tick timings, displays its report, and then resets all the tallies.

```
// ================================
// The Lively Kernel Profiler
// ================================
Object.profiler = function (object, service) {
  // Invoke as, eg, Object.profiler(Color.prototype, "start")
  var stats = {};
  var fnames = object.constructor.functionNames();

  for (var i = 0; i < fnames.length; i++) {
    var fname = fnames[i];
 if (fname == "constructor") {} // leave the constructor alone
 else if (service == "stop") // restore original functions
     object[fname] = object[fname].originalFunction;
 else if (service == "tallies")
     stats[fname] = object[fname].tally; // collect the tallies
 else if (service == "ticks")
     stats[fname] = object[fname].ticks; // collect the real-time ticks
 else if (service == "reset") {
     object[fname].tally = 0; object[fname].ticks = 0; // reset the stats
 } else if (service == "start") {
     // Replace original functions by tallyFunc wrapper
```

```
        var tallyFunc = function () {
            var tallyFunc = arguments.callee;
            tallyFunc.tally++;
            msTime = new Date().getTime();
            var result = tallyFunc.originalFunction.apply(this, arguments);
            tallyFunc.ticks += (new Date().getTime() - msTime);
            return result;
        }
        // Attach tallies, and the original function,
        // then replace the original
        if (object[fname].tally == null)
          tallyFunc.originalFunction = object[fname];
        else
          tallyFunc = object[fname]; // Repeated "start" will work as "reset"

        tallyFunc.tally = 0;
        tallyFunc.ticks = 0;
        object[fname] = tallyFunc;
 }
}
 return stats;
};
```

Listing 1.3. The Lively Kernel Profiler

The Profiler shows the degree to which a dynamic language environment can amplify even the simplest reflective capability. In this case the millisecond clock, and ability to wrap and replace methods yields a relatively powerful profiling tool in only half a page of code.

It is lamentable that the JavaScript standard provides almost no access to the runtime execution state, such as call stack, temporary variable values, and the ability to resume a suspended computation, but we can at least make the most of what is there. JavaScript does provide a pseudovariable “arguments” whose value is an array alias of the arguments passed on call. It also tacks a “callee” property onto that array that allows access to the function that is running. Is this enough for reasonable debugging? In some JavaScripts it is almost enough to provide a stack trace because a non-standard feature in some JavaScripts allows a function object to return its “caller”, but this is not a proxy to the activation record, and thus is useless in the presence of recursion.

Wrap-and-replace to the rescue! In the Lively Kernel we support a debugging mode of execution that wraps every method in the system with a function which appends a reference to the arguments array, as well as to the receiving object (“`this`”) to a shadow stack that is created afresh each time through the Morphic event loop. This allows us to provide not only a stack trace, but also the ability to inspect the receiver and arguments at every level of the call chain, either at will or when an exception is encountered. It is worth noting that this management of our own stack allows us access to these values after an exception has been thrown, whereas our experience is that most JavaScript engines discard this state before giving control to the exception handling code. It can be viewed as a tribute to the power of today's computers that this level of simulation does not bring the Lively Kernel to a complete halt. In fact we hardly notice the impact on performance.

7 Team Programming

One last tool is worthy of mention, given the context of this paper. The Lively Kernel includes a rudimentary file parser which provides a bridge between the source code file style of most Java and JavaScript developers, and the per-method management of source code such as we know from Squeak and similar systems. When viewing the system source files, each method has an associated source-CodeDescriptor that delimits its location in the file. We keep a careful reckoning of changes for each file so that descriptors from earlier versions of a given file can still be used to make changes in later versions (we re-read the segment as a check before committing any change). This enables our source code browser to browse both the running code in the system and the shared sources in a repository. The source code file approach is useful for team programming, given the existence of external tools such as CVS, Subversion, and the like.

We talk here of files, but the Lively Kernel, being Web-borne software, makes no reference to disk files on the user's machine. Instead we use a basic WebDAV protocol (see `http://tools.ietf.org/html/rfc2518`) that allows Web sites to be treated as read/write file systems. Of course a user may run a local file server on his laptop in order to work away from the Internet, but this style of access to resources ensures that the Lively Kernel can be used anywhere on the Web.

These source code objects are useful in a number of different ways. For instance, one can type a search string in the Lively Kernel, and get an instant list of all occurrences in the source database that match the string. Such search results are presented as a change list viewer, and the methods so viewed can be edited there in place. It is a gratifying result of the Lively Kernel's compactness that these searches scan our entire code base and display their results in just one second.

Associated with each world in the Lively Kernel is a list of changes that have been made to the system. These can be viewed as a change list, which is handy both for viewing prior versions, and for ready access to just the work in progress. Most importantly, this log of changes is retained within the running system. If the user saves any world as a new Web page, the associated list of changes for that world will also be saved. At a later time, on a different machine, in another browser, that page can be reloaded, the change list replayed, and the work continued in a seamless manner. In this way, the Lively Kernel enables a Smalltalk-image style of evolutionary development.

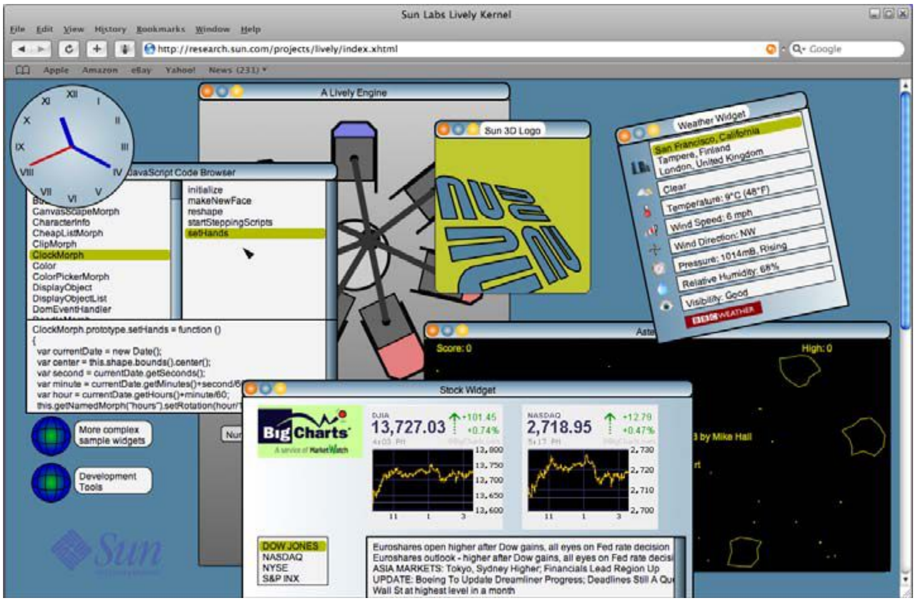
8 Application Development for the Web

Is a system capable of self-support necessarily a good foundation for the development of general-purpose applications? We believe so. We have experimented with simple media, interactive games, RSS feeds, chats, and mashups and, in every case, the architectural substrate of the Lively Kernel has shown itself to be effective. Some early experience is recorded in our Sun Tech Report [1].

Figure 8 shows a Lively Web page that is a mashup of a number of applications, all active and all manipulable. The clock and browser will be familiar from earlier in this paper. The other applications include an asteroid-blasting game, a Web weather viewer, a Web stock report, a demonstration 3-D viewer, and a simulation of a seven-cylinder radial engine (running). Also visible are a couple of links to other worlds with more applications including an RSS feed reader, a GoogleMaps viewer, and a personal information manager.

When the Lively Kernel stores a simple object, an application, or an entire world, it does so on a Web page. If one looks at one of these pages, one sees a link to import the core of the Lively Kernel, followed by markup describing the objects that have been stored. The link to the Lively Kernel both declares this to be a page of lively content, and provides the interpretive engine for bringing the stored objects to life.

Of course, similar things have been done on the Web for years, using plugins and applets to provide the engines of active content. The twist in the Lively Kernel is to use JavaScript for all the machinery of activity, thus avoiding the need to install a plugin. Other libraries, such as Dojo or Scriptaculous, can operate without installation, but the Lively Kernel goes several steps further. First, since its graphical library is built from the ground up in JavaScript, it sets the stage for a world without HTML and the epicycles that revolve around it.


Sun Labs Lively Kernel
File Edit View History Bookmarks Window Help
http://research.sun.com/projects/lively/index.xhtml
Google
Apple Amazon eBay Yahoo! News (231)
A Lively Engine
Sun 3D Logo
Weather Widget
JavaScript Code Browser
XII I II III IV V VI VII VIII IX X XI
More complex sample widgets
Development Tools
Stock Widget
Score: 0
High: 0
BBC WEATHER

CanvasScapeMorph
CharacterInfo
CheapListMorph
ClipMorph
ClockMorph
Color
ColorPickerMorph
DisplayObject
DisplayObjectList
DomEventHandler

initialize
makeNewFace
reshape
startSteppingScripts
setHands

```
ClockMorph.prototype.setHands = function ()
{
  var currentDate = new Date();
  var center = this.shape.bounds().center();
  var second = currentDate.getSeconds();
  var minute = currentDate.getMinutes()+second/6
  var hour = currentDate.getHours()+minute/60;
  this.getNamedMorph("hours").setRotation(hour/1
```

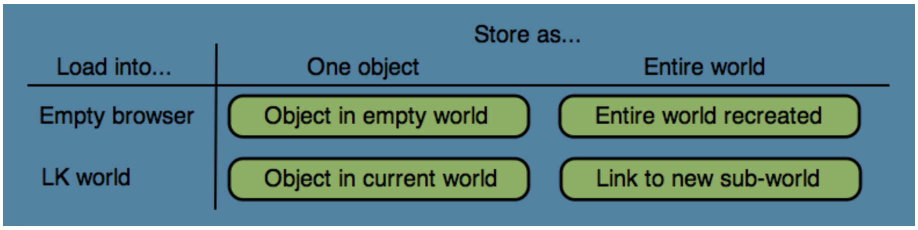
San Francisco, California
Tampere, Finland
London, United Kingdom
Clear
Temperature: 9°C (48°F)
Wind Speed: 6 mph
Wind Direction: NW
Pressure: 1014mB, Rising
Relative Humidity: 68%
Visibility: Good

DJIA 13,727.03 +101.45 +0.74%
NASDAQ 2,718.95 +12.79 +0.47%

DOW JONES
NASDAQ
NYSE
S&P INX

Euroshares open higher after Dow gains, all eyes on Fed rate decision
Euroshares outlook - higher after Dow gains, all eyes on Fed rate decisi
ASIA MARKETS: Tokyo, Sydney Higher; Financials Lead Region Up
UPDATE: Boeing To Update Dreamliner Progress; Deadlines Still A Qu
Wall St at highest level in a month

Fig. 8. The Lively Kernel running numerous applications in an overlapping window framework. This is a Web page.



Load into...	Store as... One object	Store as... Entire world
Empty browser	Object in empty world	Entire world recreated
LK world	Object in current world	Link to new sub-world

Fig. 9. Modes of storing and retrieving Lively Kernel objects

Second, it brings with it a world model in which everything is active and reflective from the beginning, a world of concrete manipulation that is immediately empowering to developers and users alike.

Figure 9 illustrates the simple modes of storing and retrieving Lively objects on a Web page.

9 A Benchmark Kernel

Beyond its immediate utility, the simplicity and completeness of the Lively Kernel make it a meaningful benchmark of system complexity. It provides a widget set and tools to construct an application from those widgets. It provides screen management and process management facilities, along with a modest IDE. In short, it provides all the tools needed for application development and deployment, as well as for evolution of the system itself.

We consider this accumulated functionality to be a meaningful unit of comparison. How many lines of code does it take to produce such a kernel? How is the performance, and by what is it most limited? These and other questions can be asked, and answered concretely with an artifact like the Lively Kernel as a benchmark.

The reader may be interested to know our experience so far in this regard. Figure 10 presents a breakdown of the Lively Kernel by functional category in terms of lines of code.

The figures above include comments and lines with single bracket characters. Prototype.js is an open source set of useful JavaScript extensions, of which we use only a small number (see `http://www.prototypejs.org/`), and JSON is a nice encoding for JavaScript objects by Doug Crockford (see `http://www.JSON.org/`).

It is surprising to some that text should be the largest module in a kernel such as ours. In our experience, it is often the handling of text that makes the difference between a toy and a serious tool. The Lively Kernel text is built from the ground up, so it can run where no native text support is available. The tally includes all the functions for mouse tracking, line composition, selection, rich text encoding, font changes, and on-the-fly layout. It seemed to us the only approach consistent with a world of active objects.

Clearly this system goes beyond the minimum functionality required for self-support. Windows, nested worlds, rich text, arbitrary scaling and rotation are

319	Host Interface - Browser API plus XML utility functions
515	Utility - Classes, collections, printing, etc. (plus 562 in Prototype.js)
833	Basic Graphics - Point, Rectangle, Transform, Color, Gradient, Image
550	Shapes - Host graphical objects (SVG node API)
1667	Morph - Basic protocol
1170	Morphic Core - World, Hand, Event, Handles, Handlers
1188	Basic Widgets - Button, List, Menu, Dialog, Slider, Selection, ImageMorph
1921	Text - basic TextMorph plus composition and rich text
747	Editors - Drag/drop manipulation, shape editors and text editing also ColorPicker and StylePanel
386	Model, Widget
1295	High Level Widgets - Scroll panes, panels, windows, world links also Panel and Browser support
1311	Tools - Browser, inspector, stack viewer, change lists, profiler
406	Serialization - Copier, exporter, importer (plus 202 in JSON)
281	Network - URL, HTTP, WebDAV basics
365	Storage - WebDAV, file browser
12954	Total

Fig. 10. Breakdown of Lively Kernel by function with approximate code size

all in some sense frills. Our intention in carrying the work this far was to make it more likely that people might pick up the work and do surprising things without needing to build a lot more infrastructure.

As with other benchmarks, we see the Lively Kernel as a starting point The challenge is to build an even simpler graphical model, an even more general processing model, a smaller complete kernel, and so on. The standard to be met, in every case, is a kernel capable of building itself, and altering and saving itself again as a Web page.

10 Related Work

It will be obvious to most readers that the Lively Kernel inherits much genetic material from the Squeak Smalltalk system and the Smalltalks that preceded it as well. Also the Morphic architecture, while most directly inherited from the (class-based) Squeak version, began as part of the Self project at Sun.

We know of no other self-supporting JavaScript development system, let alone one that runs directly off a web page without installation. However numerous web sites use the underlying JavaScript engine to provide an execution facility for JavaScript snippets, usually as part of a tutorial environment. An especially nice one is Takashi Yamamiya's live JavaScript Wiki (`http://metatoys.org/propella/js/workspace.cgi/Home`). Interesting examples of other JavaScript-hosted execution environments include "LogoWiki" by Colin Putney, Avi Bryant and Alan Kay, and an OMeta-based Smalltalk page (`http://www.cs.ucla.edu/~awarth/ometa/ometa-js/`). In this regard, Alex Warth's OMeta system (see ref) is relevant, as it facilitates this kind of emulation in the Web environment.

The Lively Kernel project at Sun Microsystems has many ties to work at Viewpoints Research, best summarized in two technical reports (see refs). Much more thorough coverage of this related work and others may be found in these reports.

We have learned much about JavaScript as a programming language. While it is beyond the scope of this paper, some early impressions are recorded in a Tech Report [11].

11 Future Work

This kernel we have described is lively in yet one other respect: It is small and simple, and thus easy to change and port. So if we imagine a secure subset of JavaScript, or a nice 3D graphics system for the Web, it should be straightforward to port the Lively Kernel to these environments and to observe immediate results in the practical world of active Internet content. We have already gone through several complete rewrites of the system and have found it to be a tractable task. For instance we rewrote the entire graphics substrate from one built on a flat graphics model (Java2D) to a retained graphics model (SVG) in roughly two months. Our experience suggests that many meaningful transformations of the Lively Kernel could be done by a graduate student or other serious programmer in a month or two. Simple experiments can, of course, be tried in much less time.

We list here some areas that we have identified for future work:

Caja. [`http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf`] is a secure subset of JavaScript. At the time of this writing, it is a research project that has not yet been tried on real applications. But if the Lively Kernel can be ported to the Caja model, it will be an existence proof of an entire application platform with known modularity and security properties. We hope to produce a version of the Lively Kernel that is consistent with the Caja rules for security, thus ensuring that Lively mashups and other cooperating applications will be well-behaved.

Lessphic. [`http://piumarta.com/software/cola/canvas.pdf`], as its name suggests, is an alternative to the Morphic architecture with various desirable properties. We are investigating a port of the Lively Kernel to the Lessphic model for the purpose of validating some of the apparent benefits of this design [10].

GUI Builder for the Internet [`http://users.ipa.net/~dwighth/smalltalk/Fabrik/Fabrik.html`]. The current Model and Widget framework of the Lively Kernel has been designed to facilitate extremely simple (drag-and-drop) construction of useful panels to control all sorts of Web-based resources. We hope to demonstrate a number of these in the future [14].

End-user programming. All of the required elements to support an Etoy-like environment already exist in the Lively Kernel. We believe that could enable the creation of interesting active Web objects conceived and built by end users.

Beyond JavaScript. [`http://piumarta.com/software/cola/canvas.pdf`]. We have only used JavaScript because it is available in every browser. We find that we have no need for a number of features in the language, and this suggests the possibility of simpler host implementations that are smaller and run faster [9].

Beyond SVG. [`http://www.vpri.org/pdf/steps_TR-2007-008.pdf`]. We have similarly been using SVG because it is available in many browsers. Here again, we find no need for many features in the standard, and this suggests the possibility of simpler implementations that are smaller and run faster [7].

Beyond Browsers. Having turned Web programming upside down in order to achieve a simpler and more general world within the browser it is hard not to ponder, from time to time, going all the way and building a complete browser within the Lively Kernel.

12 Conclusion

During the Lively Kernel project, we have learned some things about the kernel as a concept and as a vehicle. Rather than complain about the languages available or the inconsistencies between various browsers, we have done our best to pick one viable solution and to preserve every bit of liveliness for the developer and ultimately for end users. Rather than dwell on perfection in one area or another, we have pressed for the ability of end users to immediately publish and share their their creations. Having glimpsed the possibility, our passion is now to enable such authoring and sharing for every user of the Internet. It is our hope that, seen in this fresh perspective, and now available as a tangible artifact, the Lively Kernel may inspire further progress toward simplicity, generality and liveliness in Web programming.

Acknowledgements

The authors wish to acknowledge the help of Charles Jackson, Alan Lancendorfer, and Mary Holzer for their help in setting up the Lively Kernel Web site, and Pekka Reijula and Mikko Kuusipalo for their contributions as interns, including much early testing of the Lively Kernel application framework. Also Richard Ortiz for his help with a number of application and porting experiments, Kristen MacIntyre for contributions to text display and affine transforms, and Mario Wolczko, Bob Sproull and Greg Papadopoulos for their enthusiastic support of this project.

References

1. Taivalsaari, A., Mikkonen, T., Ingalls, D., Palacz, K.: Web Browser as an Application Platform: The Lively Kernel Experience. Sun Microsystems Laboratories Technical Report TR-2008-175 (January 2008), http://research.sun.com/techrep/2008/abstract-175.html
2. Ungar, D., Smith, R.: SELF: The Power of Simplicity. In: ACM SIGPLAN Notices (December 1987)
3. Maloney, J.H., Smith, R.B.: Directness and liveness in the Morphic user interface construction environment. In: Proceedings of the 8th annual ACM Symposium on User Interface and Software Technology (UIST), Pittsburgh, Pennsylvania, pp. 21–28 (1995)
4. Maloney, J.H.: Morphic: The Self User Interface Framework. Self 4.0 Release Documentation. Sun Microsystems Laboratories (1995)
5. Ingalls, D., Kaehler, T., Maloney, J.H., Wallace, S., Kay, A.: Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In: The OOPSLA 1997 Conference (1997), http://ftp.squeak.org/docs/OOPSLA.Squeak.html
6. Kay, A., et al.: STEPS Toward The Reinvention Of Programming, http://www.vpri.org/pdf/NSF_prop_RN-2006-002.pdf
7. Kay, A., et al.: STEPS Project First Year Report (2007), http://www.vpri.org/pdf/steps_TR-2007-008.pdf
8. Miller, M., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized JavaScript (January 2008), http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf
9. Piumarta, I.: COLA whitepaper: Albert, VPRI Research Note RN-2006-001-a, http://vpri.org/pdf/colas_wp_RN-2006-001-a.pdf
10. Piumarta, I.: Lessphic: A disposable, light-weight graphical enviroment for FoNC, http://piumarta.com/software/cola/canvas.pdf
11. Mikkonen, T. and Taivalsaari, A.: Using JavaScript as a Real Programming Language. Sun Microsystems Laboratories Technical Report TR-2007-168 (October 2007), http://research.sun.com/techrep/2007/abstract-168.html
12. Various, History of Morphic, http://wiki.squeak.org/squeak/2139
13. Warth, A., Piumarta, I.: OMeta: an object-oriented language for pattern matching. In: Proceedings of the ACM 2007 Symposium on Dynamic languages, pp. 11–19 (2007), http://portal.acm.org/citation.cfm?id=1297081.1297086
14. Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., Doyle, K.: Fabrik: A Visual Programming Environment. In: Proceedings of the ACM OOPSLA 1988 conf., pp. 176–190 (September 1988)

On Sustaining Self

Richard P. Gabriel

IBM Research
rpg@dreamsongs.com, rpg@us.ibm.com

There are three approaches, I think, to sustaining the health of running systems: designed perfection, instinctual adaptation, and learning. For small programs-and for increasingly large programs over time-designed perfection is a viable approach. Such software has been proved (rigorously) or rationally has come to be believed to be impervious to variations in its environment, which is itself assumed reliable using some mechanism. Software with designed perfection is typically the most efficient, because it does not waste resources on adaptation or learning. Experience has shown there is very little such software, but hopes remain high and research is active and fruitful.

Instinctual adaptation uses built-in adaptation mechanisms to handle changes in the environment. Sensors and effectors can be used to gauge the environment-including the internal environment of the software-and make changes to the software so that it can perform well. With the use of feedback mechanisms and even evolution over time, entities that adapt instinctively to their environments can demonstrate remarkable adaptability and survivability. Spawning copies to handle larger loads, migrating nearer to frequently and heavily used resources, and seeking collaborating services are examples of activities software components can do that are instinctual. All the tools and mechanisms of of complex adaptive systems are in play here: swarming, ant colony optimization, metaheuristics, and game theory-all examples and not an exhaustive list-can all be applied here. Systems that use this approach incur an overhead, both of space and time: sensors needs to operate, some processing must be done to determine the right action to take, and the effectors need to work. This is all in addition to what the software has to do anyway. This approach works best when the environment is complex and changes-but probably only medium complexity and slow changes.

Learning requires perception-representation-action loops within the software to enable it to learn how to adapt and use the environment. Think about neural nets, Bayesian classification, and reinforcement learning as examples. In such systems, the environment (including perhaps the internal environment) is sensed, a representation is made, and action is taken based on "thinking about" those representations. The representations and the "thinking about" processes are altered based on the outcomes of the actions-this is what makes this a learning approach rather than instinctual adaptation. The overhead for learning is even greater than for instinctual adaptation. Not only do the sensors have to sense, the processing of that data has to take place, and the effectors have to work, but work needs to be done to improve the intermediate processing (which is the learning). And what's worse, before learning takes place, the software that needs to actually do something can be behaving poorly and probably incorrectly-that is, it's likely making mistakes. But when the environment is complex, is changing rapidly and perhaps dramatically, and the environment includes other learning entities, this is the best approach.

Therefore, there are benefits and costs to using learning over instinct as well as to using instinct over designed perfection. In the natural world, designed perfection is mostly the realm of inorganic matter. Such things don't improve or adapt, only change through deterioration. Instinctual adaptation is the realm of all living entities. And any being with a nervous system is able to learn, probably, though this is not thoroughly tested. What is interesting is that the few experiments that have been done have shown that in the wild, the ability to learn is not fully exploited-some flies, for example, are capable of becoming more accomplished learners, but have not; and the reason is that being better learners and even merely engaging in learning have costs in terms of survival rates in the wild [1].

Adaptable approaches to health have some requirements. First is that it be possible to see into the environment that the entity is sustaining itself within, including, where necessary, its own internal mechanisms. Immune systems require the ability to see into a cell to determine whether it is a healthy member of "self." Without this, resistance to disease and other attackers would be left to barriers and the other, cruder supporting members of the immune-system cast.

Second is that there must be mechanisms that can take action and make changes. For example, in chemotaxis, feedback loops control the methylation and demethylation of transmembrane receptors; this (and some other mechanisms such as receptor clustering and receptor-receptor interactions) enables cells to be able react to a wide range of chemical concentration gradients, as if "remembering" what has recently been "seen," and thereby move toward food sources and away from toxins. Methylation and demethylation are effector mechanisms in this system.

Next, perfect execution must not be required. Adaptation means moving from a less perfect to a more perfect configuration for the conditions at hand, which means that a range of not-fully-working behaviors must be able to be tolerated.

Next, complete before-execution design and implementation must be impossible or impractical. That is, if designed perfection is an option, why not use it?

And finally, the combination of sensing, possibly processing, and then acting must represent a reasonable if not total model of how to interact with the environment, including the internal environment. Another way of putting it is that the entity that is sustaining itself must be an effective model for itself.

Evolution can be viewed as a form of learning at the population level, but I regard it as a meta-level mechanism for changing the operating design and parameters for the population. Thus, for example, evolution can change the instinctual mechanisms or improve (or degrade) the ability to learn (quickly) in order to balance the costs and benefits of learning.

Now notice this: all the requirements for adaptable entities are contrary to the requirements for designed perfection based on evidence provided by the work and recommendations of programming language and software engineering researchers.

To achieve designed perfection, visibility into the environment is limited to parameters that are passed (one way or another), with global information frowned upon; visibility into modules is even more strictly discouraged-and in many programming languages it's impossible. Similarly, the ability to effect change at runtime is either frowned on, forbidden, or not possible. The reason for these restrictions is that they enable being able to reason about and eventually prove that certain errors are not possible-this is what type systems are mostly about (that and execution efficiency). Moreover, the languages whose programs are easiest to prove things about are those without side effects.

A few researchers are starting to look at imperfect execution as a viable alternative to perfection. Many mainstream researchers who hear their ideas are shocked-sometimes they laugh. The notion that perfect execution might not be required is contrary to everything they've ever learned. Designed perfection is predicated on being able to do a perfect static design, either primarily with people doing the designs or people creating models from which provably correct code is produced. Therefore the situation of it being impossible or impractical to do is assumed away from the get go.

Finally, though the source code is considered by today's researchers as the true model for the software-hence the word "static," which means roughly "textual"-the dynamic / runtime model is generally considered distasteful though more minutely accurate than the source, and languages with the right level of reflection capabilities are not so common as would be preferred. "Distasteful" only because it is difficult to know whether the runtime samples you are able to gather represent all the possible states (up to equivalence), and hence whether you know everything about the program.

Although the requirements for designed perfection and the adaptive approaches are diametrically opposed, I don't want to leave the impression that one set is good and the other bad, or that designed perfection should be abandoned. If it's possible to make some part of a software system perfect, please do it. I am confident that over time, the size and complexity of systems that can achieve designed perfection will grow, though more slowly than systems that don't aspire so high.

The point is that roughly all programming language and software engineering researchers are pursuing designed perfection, and we need some who are looking at the adaptive approaches to keeping software systems alive, well, and functioning acceptably.

Reference

1. Mery, F., Kawecki, T.J.: An operating cost of learning in Drosophila melanogaster. Animal Behavior (2004)

Huemul – A Smalltalk Implementation

Guillermo Adrián Molina

huemul@losmolina.com.ar
http://www.guillermomolina.com.ar/huemul/

Abstract. Huemul is a new implementation of Smalltalk. It is built under the principle of reuse of existing technologies. It aims to be compatible with Smalltalk 80 at the language level. Huemul does not interpret Smalltalk code. It translates methods directly to machine code, and they are kept like that in the image. With this approach, virtual machine code is kept to the bare minimum.

Keywords: Smalltalk, virtual machine, JIT.

1 Introduction

When Smalltalk was first introduced, Operating Systems lacked many services and features needed by applications. The set of tools they provided were rather small, every application had to do everything by itself. Instead of focusing on the core business, an application programmer had to provide everything from graphical interfaces to multitasking, file support, etc. Smalltalk was not an exception to that rule.

But Smalltalk wasn't a normal application, it was a complete graphical environment that complemented the lack of functionality of the underlaying Operating System. The Smalltalk user (or developer) didn't have to worry about lower levels, because they were provided by the Smalltalk environment. One would just concentrate on the important, by reusing what was already made.

As Operating Systems became more and more powerful, applications tended to trust many of their basic functionality to them. It is unlikely that a new application developed today would implement a communication or graphical framework of its own. As all the applications shared their need for basic behavior, most of the functionality was standardized, and concentrated in libraries. On top of the libraries, extended behavior formed frameworks, which could be used by the application programmer, in the language of his choice. The result of this process led to the model in Figure 1.2a.

Almost thirty years has passed since the introduction of Smalltalk. Smalltalk systems have evolved too, and there are now many implementations. Some of them focus on speed, others focus on usability, others on simplicity, etc. But the basic underlying structure of them still resembles the original one.

The current tendency is to delegate basic functionality to the Operating System, while maintaining some sort of control of the lower infrastructure and adapt it to the upper layers. Figure 1.2.b shows how actual Smalltalks interface with Operating Systems.

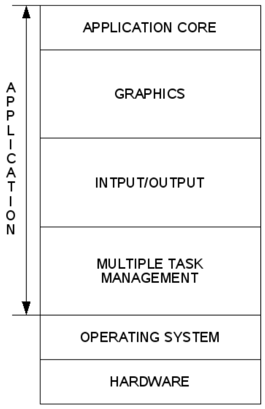


Fig 1.1a.

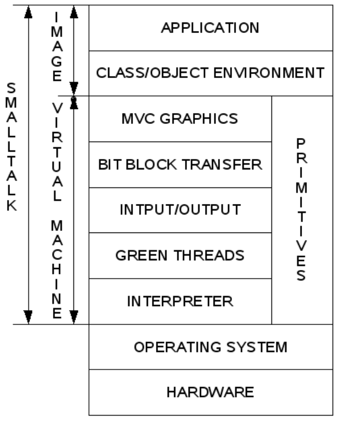


Fig 1.1b.

Within these actual implementations, Squeak [2] is the implementation that resembles most the first Smalltalk System. It still draws its own interface, and interprets its own code, manages its own processes, with some help from the Operating System.

An advanced Smalltalk System like Visual Works [4] implements internally an interface to the underlaying Operating System, it also interprets its code, but optimizes its execution on the fly, thanks to some advanced dynamic compiling and execution features.

Dolphin Smalltalk [5] takes a different approach. It tries to use as much native functionality as possible. From native threads, to native graphics system. But it still uses an interpreter layer.

There is a Smalltalk implementation called Smalltalk/MT [15] that can generate applications that doesn't need an interpreter to run. But the developing processes is

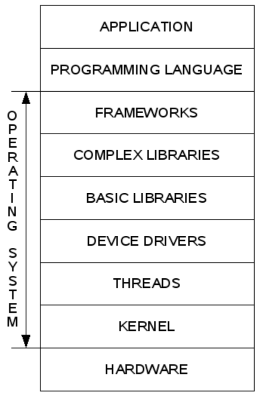


Fig 2.2a.

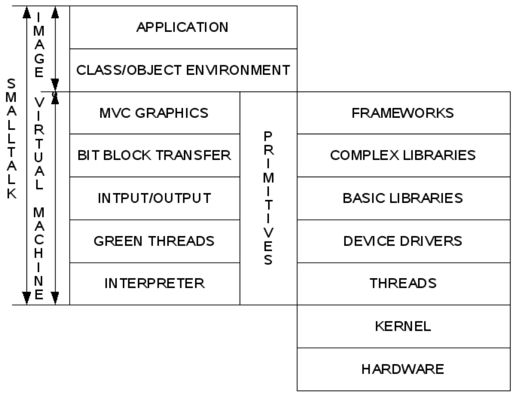


Fig 2.2b.

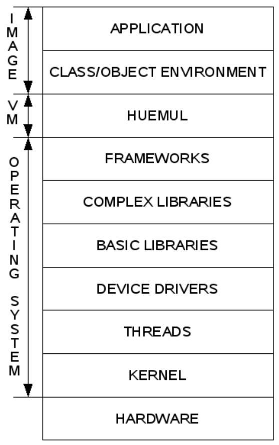


Fig 2.3.

made by a normal interpreted Smalltalk, and as a last step before production, a native code executable can be generated. This executable once generated, becomes a stand-alone application. It is neither a self sustained system nor a part of it.

But what can be observed in these three implementations is that all of them include a certain amount of coding in their Virtual Machine that is already present in the system, the interpreter is an example of that duplicity.

Resources are scarce. Why don't we use those resources to get further on one path, instead on splitting those resources in parallel paths that lead to the same place?

In this paper we present Huemul. Huemul is a new Smalltalk implementation. With Huemul, we take the concept of reusability to an extreme. Instead of providing its own libraries, Huemul tries to delegate as much as possible to the Operating System. If it is already there, Huemul just uses it.

As can be seen in Figure 2.3, Huemul's Virtual Machine is tiny. It has just the minimum necessary to bootstrap the image, connect to the Operating System libraries and set up the dynamic call infrastructure. Huemul uses native code, native threads, native widget framework, native input/output, etc. It provides mechanisms to add new primitive functionality and access dynamic libraries on the fly, without touching the Virtual Machine at all, or creating any non-Smalltalk plugin.

Huemul does not even have an interpreter. Every time a method is created or modified on the running system, it is compiled to native code, and it is immediately available for use. When the image is saved, all the methods are saved with it. Compiled methods are normal objects like any other object in the system. They can be accessed, modified or deleted dynamically.

In the following chapters we are going to introduce some of the details of the implementation of Huemul.

2 Virtual Machine

Virtual Machines [18] (VMs) provide a platform-independent programming environment that virtualizes details of the underlying hardware or operating system, and allows a program to execute in the same way on any platform. The size and complexity of the VMs varies, but the minimum abstraction mechanism may include at least: the CPU, memory, peripherals and support for the programming language.

The processor is the main abstraction that a VM should include. The processor virtualization layer is implemented by an emulation engine. The interpreter is in charge of executing platform-independent code, in a platform-dependent environment.

Most of the time, the illusion of having more than one unit of execution is handled by some sort of multiplexing mechanism implemented at the VM level. This mechanism is called green threads as opposed to the platform-dependent mechanism called native threads. Green threads are lighter than native threads, and that is why they are called green. But what is most interesting about green threads, is that the VM has total control of them.

VMs also virtualize memory. The allocation mechanism is always handled by some sort of Object Memory. This Object Memory replaces the standard library function malloc. The memory allocation is extended by this means to suite the needs of the virtual architecture that is being implemented. The restoring mechanism of memory is handled by some sort of Garbage Collector. The Garbage Collector automattes the the return of memory to the free pool.

The peripherals are implemented differently depending on the VM architecture, and also on the programming language that the VM may support. In Smalltalk, primitives are the link provided at the language level to support low level extensions. At the VM level, these primitives are plugged into VM extensions that are either embedded in the VM, or implemented by some plugin mechanism or in the form of a dynamic library.

The support to the language is the mechanism that each VM has to provide to present the run-time environment to the user. In Smalltalk, the VM should include the handling of exceptions, method binding, closures, contexts, etc.

Huemul tries to free the VM from all the complexities, and either tries to push them down to the underlaying Operating System or pull them up to the image. With this methodology, Huemul's VM is just 4500 lines of code, and has an executable of less than 100KB. Huemul does not have an interpreter. Most of the extensions are driven by OS libraries with practically no VM intervention. Huemul uses native threads instead of green threads. The implementation details are explained in the following chapters.

3 The Interpreter

In many Smalltalk implementations, the first execution phase is handled by the interpreter inside a classic Virtual Machine (VM) that reads in and executes the bytecodes. Think of the VM as a simulator for a complete machine, and the interpreter is a CPU simulator whose instruction set is Smalltalk bytecodes. However, interpreting bytecodes makes a Smalltalk program several times slower than comparable C or C++ programs [13].

Smalltalk is a dynamic language, modification of the code at runtime is not only allowed, it is also encouraged. To implement this behavior, every time a method is created or modified, it has to be translated to bytecodes on the fly. This work is done by the Smalltalk compiler. The compiler has two phases, the first one is done by the parser. The parser translates source code to the Abstract Syntax Tree (AST). The AST is a more convenient computer-usable representation of the code. The second phase is done by the translator. The translator converts the AST to bytecodes. When a message is sent to an object, one of the methods is selected by the VM, then its bytecodes are loaded and executed by the interpreter.

Using an interpreter adds quite a bit of execution overhead. A common solution for high-performance VMs is to use dynamic translation implemented in Just In Time (JIT) compilers.

Most of the VMs that include JIT compilers still have to do the procedure explained above. JIT mechanisms are applied at runtime. When the runtime system notices a method has been called enough times to make it worthwhile to generate machine code, it is translated on the fly. Future calls to the method will execute the machine code directly. Think of the code generated by the JIT compiler as a method code cache that simply run faster than interpreted code. This can provide certain speed-up, but better results are obtained with a different procedure.

A further step can be taken to achieve higher performance. Object oriented programming languages don't gain much execution speed by implementing classical static optimization techniques [13]. A dynamic gradual optimization based on live statistics taken from the running code gives a much better result. These advanced techniques include algorithms like Polymorphic Inline Caches (PICs), type feedback, aggressive dynamic code inlining, etc [13, 19]. With these techniques, instead of trying to generate the fastest code at the first compilation, JIT compilers gradually produce faster compiled code, as the statistical information becomes richer. In this way, highly optimized code is used to execute the most called methods, which leads to a high speed-up in the overall execution of an application. The continuous compilation of code has the disadvantage of being resource intensive, but the result clearly outperforms the effort of doing so.

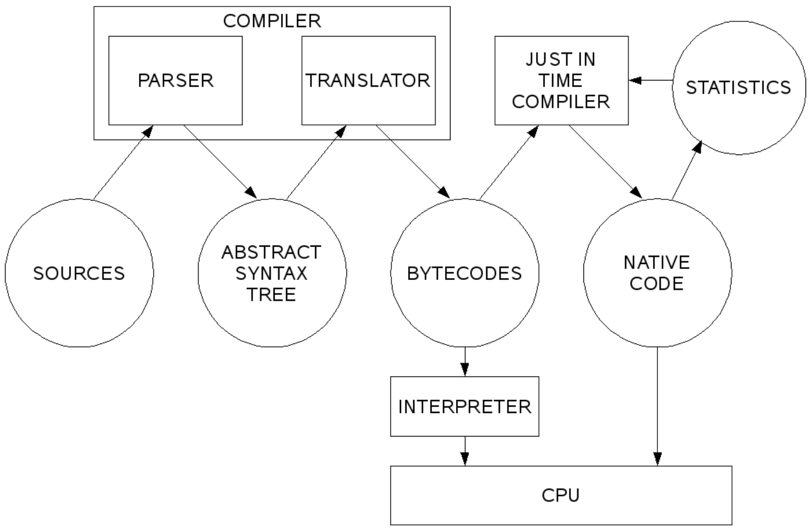


Fig 3.1.

Interpreters are fairly complex pieces of code. A big part of a VM is devoted to the interpreter. There is a development cost associated with them, as they have to be coded, maintained and debugged. Also a good bytecode instruction set has to be chosen. Compiler translators are complex pieces of code too. They have to be compatible with the bytecode instruction set that has been chosen, and they have to be coded, maintained and debugged too.

As can be seen in Figure 3.1, and from what has been explained above, the translator-bytecodes-interpreter-CPU phase, resembles a lot in terms of functionality to the JIT compiler-native code-CPU phase. If we could just get rid of one of these two phases, we would cut the effort of developing the VM substantially. We would have to maintain just one compiler, instead of two. We wouldn't have to implement a bytecode instruction set. And what is most important, we wouldn't need an interpreter at all. As the generated code is native from the very first compilation, the CPU is the device used to interpret, or in this case, just execute the code.

This is where the reusability philosophy enters. Huemul uses the CPU as a "hardware interpreter", and uses the native machine instructions of the CPU as bytecodes, saving significant VM space and development effort. It also benefits from the fact that the first (and almost unoptimized) execution will be done faster than if it would have been interpreted. Another advantage of this approach is that it has no development effort in the interpreter, as the device that we use is already there, we just reuse it. And last of all, Huemul could still benefit from most of the techniques available to speed up dynamic object oriented languages, like PIC and automatic recompilation and inlining of code, because these phases are implemented later in the execution phase.

Huemul's compiler is implemented totally in Smalltalk. Both the parser and the translator are ported from Squeak. By using the same parser as Squeak we ensure a complete compatibility at the language level to it. Exupery [10] is a JITer for Squeak.

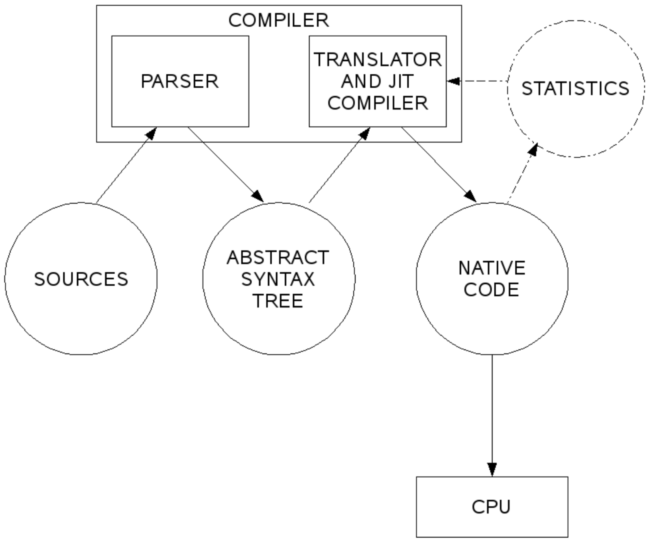


Fig 3.2.

It translates from Squeak bytecodes to native x86-32 machine code, and also performs dynamic optimizations. We ported Exupery to Huemul as its translator from AST to machine code.

Exupery works roughly in three phases, the first one generates high level AST from bytecodes, the second one transforms high level AST to low level AST, and the third one generates machine code.

In Huemul, we generate the AST nodes from Squeak's new compiler parser [3]. This AST is neither compatible with Exupery's high level AST nor with the low level one. So we developed a transformation from the Squeak AST to the low level Exupery AST. We then we generate machine code with the third pass of Exupery. We don't use any of the dynamic optimizations built in in Exupery, but we are planning to implement them sometime in the future.

The execution environment of Huemul is represented in Figure 3.2. Huemul tries to overcome the lack of dynamic optimization by statically inlining the most frequently used methods, like integer operations and general methods like #at:, #at:put, #class, etc. When Huemul's JIT extension become available, those static optimizations may not be needed any more, as they would be optimized dynamically.

We are pleased to say that in spite of this lack of functionality, Huemul still achieves reasonable overall performance. We made a superficial comparison between Huemul, Visual Works and Squeak (with and without Exupery) using Squeak's tiny benchmarks on each Smalltalk. These benchmarks measure the number of bytecodes executed per second (b/s), and the number of message sends accomplished each second (s/s). The results of these benchmarks are unfortunately not representative of how a complex application would behave. There is also a generalization problem, as the platforms benchmarked don't use the same bytecode set. Also, in the platforms with a JIT and in Huemul, there is no bytecode execution at all. Moreover, we can't even talk about how many sends are performed each second in inlining environments. But it serves as a basis to understand the point explained here.

The machine we used for the benchmark was an AMD Sempron of 3.1Ghz with 2GB of memory. Windows tests where done with Windows XP SP2, and Linux tests where done with OpenSuse 10.3, kernel version 2.6.22.17-0.1. The next table summarizes the results:

Smalltalk	bytecodes / second (b/s)	sends / second (s/s)
Squeak 3.9	93,362,509	4,250,358
Squeak 3.9 + Exupery 0.11	785,878,741	12,771,577
Visual Works 7.5 non commercial	755,162,241	80,312,362
Huemul 0.6	832,520,325	33,552,522
Smalltalk MT 5.20, evaluation version	898,245,614	25,831,057
Dolphin Smalltalk X6 2006.6.02.3 Community Version	212,096,106	10,350,066

The first four benchmarks where done in Linux and the last two in Windows. The tests confirmed what we expected with Squeak and Dolphin being the slowest platforms. It is the logical result as both of them are interpreted. On the other hand, it is good to know that Exupery makes and excellent job as Squeak's JIT, even outperforming Visual Works in the b/s benchmark. Although Huemul implements Exupery, it doesn't use it in the same way as Squeak does. And, as we have not yet implemented Exupery's optimizations, we have carefully chosen the right methods to inline statically. We also have a simple Object Memory design (explained later) that is optimized for speed. This all takes us to the second place in the b/s benchmark. Smalltalk MT is the winner in the b/s benchmark, and is compiled to native code the same way that Huemul is.

In the s/s benchmark, Visual Works clearly outperforms the rest of the implementations. These high s/s numbers are due to its effective JIT. All that this benchmark does is to send the same message to the same kind of object again and again. A good JIT would detect that behavior and would inline as much code as possible. That is why Visual Works outperforms the others. But, Huemul still outperforms Squeak even by implementing the same translator. This is because the send mechanism chosen for Huemul is simpler, and because Squeak's implementation of Exupery has to deal with its relatively complex internal architecture.

There is another tradeoff on using native code instead of bytecodes. This penalty is due to the size of the compiled methods. While most of the bytecodes are implemented with just one byte, each native code instruction requires many bytes. And most of the time, more than one instruction is required to do the same job as a single bytecode. But, code size is not the only overhead. Bytecodes are normally implemented so that the only references to the outside world are made by accessing indexed literals contained in the method itself. Huemul on the other hand, references outside objects directly within the code. Since neither the address of the generated code nor that of any designated object is fixed in memory, some relocation information is generated to support these outside references. This relocation information makes Huemul natively compiled methods bigger than normal methods.

In interpreted Smalltalks, debuggers normally create the debug information on the fly. Either by decompiling the bytecodes, or by generating AST nodes from the source code again at debug time. Unfortunately, Huemul's debugger can't use neither of the two approaches. The first approach can't be used because the code generated by the native compiler is much more complicated than bytecodes. Decompiling native code to make it match the source code is a difficult task by itself, and it is even worse if it must be done on the fly. Generating debug information from the AST while debugging is possible, but the generated code may not match the original AST nodes since a lot of optimizations are applied after the AST node generation. So the debug info generated at debug time would not match the optimized code. The solution applied in Huemul, is to generate debug info while compiling the method. This debug info is saved with the method, and it is accessed at debug time. This debug info matches the optimized code. The problem with this approach is that methods become bigger.

The following table shows a comparison between some methods in Squeak and in Huemul.

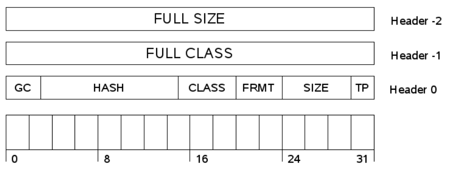
Method	Squeak 3.9	Huemul 0.5b		Ratio
		Code	Code + info	
Integer #digitDiv:neg:	598	8,027	22,691	37.94
Float #absPrintOn:base:	544	5,973	18,885	34.71
IntegerTest #testPositiveIntegerPrinting	2,939	28,236	79,252	26.96
Object #changed:	39	160	488	12.51
Class #category	80	338	1,258	15.72
True #ifFalse:	17	10	90	5.2

As can be seen, Huemul methods are bigger than Squeak methods. The ratio becomes larger as the size of the method increases. This problem makes Huemul images bigger than the images of other implementations. The average ratio of all the methods in the image is about 20 times. This is the price that has to be paid in order to use native code instead of bytecodes.

4 Object Memory

In Smalltalk, everything is an object and all the objects reside in the Object Memory. The description of the layout of each one of the objects is an important matter, as it will define how easy or difficult it is to access its information. There is a direct relationship between the steps required to access an object's contents and the overall execution speed. It will also affect the resulting size of the image. Unfortunately, we can not have at the same time, both the smallest and the fastest object layout. We have to make a compromise between the two.

Huemul inherits much of its functionality from Squeak [2]. Squeak's Object Memory is optimized for size. There are two main object layouts in Smalltalk [9]; Small Integers and the rest of the objects. This main distinction is also valid in Squeak. Allowing the Small Integers to be saved without a header, saves a lot of space, as there are a lot of Small Integers in an image. But Squeak further divides objects into three categories. Depending of the class of the object and its size, a different header is chosen. These three headers vary in size and complexity. Figure 4.1a shows the three different header types.



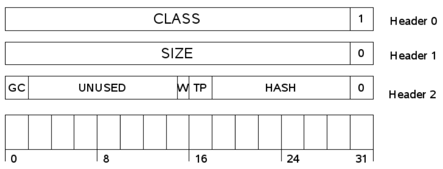


Fig 4.1a. **Fig 4.1b.**

If the object's class is one of the 32 designated (and most frequently used) classes, and its size is less than 64 words, it just gets object header 0, which is just 1 word overhead. If the object class isn't one of the 32 special classes, but it is smaller than 64 words, it gets the 2 word header. And lastly, if the object's class isn't a special class or it's bigger than 64 words, it gets the full 3 word header. This header selection procedure saves a lot of space, but on the other hand, it adds a certain amount of complexity to the basic object operations.

For example, if we need to find an object's size, we would first have to see if it is a Small Integer or not. Then we would have to see if the object is variable, then it would be allowed to ask about its size. Then, we would have to ask if the header is 1 word, or more. If the header is 1 word in length, we have to get the bits of the object that represents the size, but if the header is 3 words in length, we would also have to take care of Header -2. It is easy to see that this mechanism has much complexity. Moreover, it is fairly complex to build a good just in time compiler for Squeak, because much of the CPU time is spent in circumventing this complexity [10].

Huemul's Object Memory, on the other hand is designed with simplicity in mind. As can be seen in Figure 4.1b, every object in Huemul, except Small Integers, have the same header. This header has an overhead of 3 words. It is a fairly big overhead in terms of size, but it is a big advantage in terms of execution speed. Class, size or information access is done in a very straightforward way, and this allows an easy inlining of most of the basic object operations of the Smalltalk language.

5 Tagged Integers

Smalltalk requires that everything be an object. But, as shown in the Blue Book [9], a simple optimization for size is to encapsulate small integers. With this mechanism, we avoid using a header for Small Integers, and doing so, we gain a lot of space, since Small Integers are used a lot in a Smalltalk system. A drawback for this approach is that we have two methods of obtaining an object's class. First we have to examine if it is a Small Integer, in the case it is not a Small Integer we must look up the object's class.

The encapsulation of Small Integers, as defined in the Blue Book, is done by using the least significant bit of the word as a flag that tells if this word is a Small Integer or if it is a pointer. If it is a Small Integer, the flag is a 1, if it is a pointer, the flag is a 0, this is illustrated in Figure 5.1a.

If implemented in this way, accessing an object pointer is a direct reference, as long as every object is aligned. Small Integers, on the other hand, can't be used

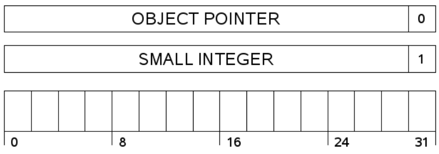


Fig 5.1a.

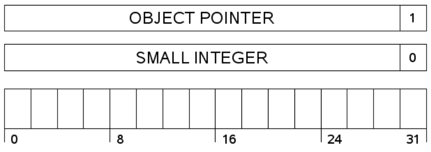


Fig 5.1b.

directly in arithmetic operations. They have to be detagged before they are used, and be tagged again afterwards. Optimizers make a good job circumventing this problem.

But there is another alternative called 0 tagged integers. As opposed to what is explained before, Small Integers have a 0 tag, and object pointers have a 1 tag, as shown in Figure 5.1.b.

With this approach, we can not access a pointer directly, because it is pointing to the next byte of where the object actually is. Now the access to the object is performed by pointing to the address minus one. It is almost the same mechanism as before, because in x86 CPU architectures, load the contents of pointer X is in fact, loading the contents of pointer X + 0. But even on architectures where the loads are not the same, the speed gain obtained from 0 tagged integer optimizations in arithmetic operations would compensate that setback.

With 0 tagged integers we don't need to detag/tag for most of the frequently used arithmetic operations. For example, the most common operation is Small Integer addition, and this can be done without the detagging/tagging mechanism. In fact addition and subtraction are the most benefited operations, since we don't have to tag or detag, and we don't even have to adjust the result to correct it.

The first version of Huemul used 1 tagged integers. When the switch was made, some changes were made to the VM and the compiler. In the VM, we had to change the macros that checked, tagged and detagged Small Integers to support the new format. But we also had to change the way we accessed to the other objects. Before the change we just used the object pointers directly, as they were normal C pointers to structures. But after the change, we had to create two new macros to tag and detag object pointers, and surround all the accesses to objects by one of these macros. This leads to having to type more, but without an execution speed tradeof. Because, when the VM is compiled with optimizations, this macros are inlined and embedded in the load instructions.

The compiler had to be changed too, in order to support the new format. But, since we already had the tagging and detagging mechanism of Small Integers, we only had to adjust it. And as mentioned above, the instructions and addressing mode to access the objects was already there, the only difference was that we had to add the -1 offset to all of them, instead of the implied 0.

When Huemul switched the use of 1 tagged integers to 0 tagged integers, the performance gain measured by the tinyBenchmarks was about a 10%. Despite the benchmarks being not very accurate, it was representative enough to demonstrate that it was worth the effort of converting the objects footprint, the changes in the virtual machine and the embedded compiler.

6 Processes

Smalltalk units of execution are defined by instances of the Process class. These Processes are most frequently implemented as application threads or green threads. This means that the Virtual Machine is in charge of the life cycle and scheduling of the processes. So most of the Smalltalk VMs have an embedded scheduler to run this mechanism.

The VM interpreters that implement green threads are written as single threaded applications. The simultaneous execution of Smalltalk processes is thereby implemented by multiplexing this single thread between all the active processes. This solution works fine for single core, single processor computers, sometimes even outperforming native threads solutions, as the creation and context switch of a green thread is cheaper than with a native thread. But single threaded applications don't use more than one CPU at a time, no matter how many cores or processes a system may have. There are some workarounds like running the interpreter, the garbage collector and other house keeping tasks in separate threads. There are also solutions that implement multiple different instances of the virtual machine running in some sort of cooperative network.

Posix threads [17] (pthreads) are the standard mechanism to implement native threads in Unix environments. Huemul uses pthreads as the back end for the Process class. In this way, the Operating System takes control of the execution and scheduling of the threads, so there is no scheduler in Huemul. A direct advantage of this behavior is the seamless and automatic integration of Huemul with with multi CPU and multi core system, as native threads are efficiently balanced among them.

7 Contexts

Every process in Squeak is further divided in smaller activation records called contexts. These contexts keep state information about the execution of methods or blocks. Instead of having one stack for each process, it is distributed among the contexts. When a method is executed, it is assigned an instance of the class MethodContext, and when a block is executed, it is assigned an instance of the class BlockContext. All the contexts have a pointer to the context in which the call was made. The block contexts also have a pointer to the context that created the block. Following these pointers we can traverse the complete stack of a process. A method context can only return to the caller, following the pointer. On the other hand, block contexts can return to the caller, or to the creator.

Huemul uses the x86-32 CPU's native code to implement this behavior. In that CPU, each process is normally bound to just one stack. When a function is about to be executed, the arguments are pushed into the stack, then a call instruction that automatically pushes the return address on the stack is issued, then the jump to the function is made. When the function is about to end, a ret instruction is issued that returns to the address previously saved in the stack. Then, the caller removes the arguments from the stack to leave it like it was before.

In Huemul, we have neither method contexts nor block contexts. We have one stack for each Smalltalk process, and we use the simple mechanism described above to implement method contexts. When a message is sent to an object, the receiver and the arguments are pushed onto the stack. Then the function that selects which method to run, given the receiver and the selector is called. And lastly, the selected method is called. When the method ends, it issues a ret instruction as described.

There is no standard way in x86-32 CPUs to unwind the stack to a place other than the immediate caller. As mentioned earlier, blocks can return to the caller, or to the creator. There is no problem to activate a block that just returns to the caller in the same way we activate methods. But if the block wants to remotely return to the creator we have to apply some mechanism to save the state of the creator. Once the state is saved, we can restore it in case of a remote return.

Huemul uses the standard setjmp/longjmp C functions to implement this behavior. Setjmp/longjmp are defined in the C standard library to provide "non-local jumps" outside of the normal function call and return sequence. The paired functions setjmp and longjmp provide this functionality through first saving the environment with setjmp to which longjmp can "jump" from a point elsewhere in the program.

If a piece of code creates a block that may remotely return, it issues a setjmp (save the environment). This helps the remote returning block to return home by issuing a longjmp (restoring the saved environment).

Apart from blocks, there is another mechanism that alters the normal flow of execution. This mechanism is triggered by the Exception Handling System (EHS). The EHS is basically supported by two methods: #ensure: and #on:do:. The first method is used when a block has to be executed whatever happens with the execution of another block. The second method is used when a block has to be executed, only when certain exception is triggered while executing another block.

Each Huemul process has a list of contexts that have been guarded against exceptions, and the blocks that have been assured. This list is maintained by the EHS mechanism. When either #ensure: or #on:do is sent to a block, the EHS mechanism issues a setjmp to save the execution state. Then the block is evaluated as usual. If the system triggers an exception, the EHS searches the list in order to find a suitable exception handler. If the correct handler is found, its exception block is evaluated, if it is not found, the default exception handler block is evaluated. If an ensured block is found during the process described above it is evaluated. The contexts of both the ensured and exception blocks is restored by a longjmp function.

8 Traits

The solution for code reusability in Smalltalk is based on single inheritance and polymorphism. But there are situations when two or more classes that don't inherit from the same root class, want to implement and reuse the same behavior, without copying the methods in all the classes. Other object oriented languages implement mixins or multiple inheritance to circumvent this problem. Traits [11] gives an alternative to those mechanisms, leading to a solution more compatible with the Smalltalk language. The idea behind a trait is to have a separated repository of methods, that can be merged in a efficient and controlled way into a class. In this way, different classes may share the same behavior, even if they don't relate to each other.

There are two major approaches for the implementation of traits; static and dynamic. With the first approach, trait compositions are flattened at compile time, and trait methods appear as normal methods to the classes that use them. With the dynamic approach, there is no physical inclusion of the trait at compile time. The binding of the method is done at runtime. While Squeak uses static traits because it doesn't require any change in Squeak's virtual machine, Huemul uses the dynamic approach. It doesn't have the complex flattening mechanism required by the static approach, instead it extends the method search mechanism, that binds the method at run time.

Huemul uses the same syntax as Squeak to operate with traits. This was made in order to get compatibility at the language level with it. Huemul's Class Browser tool is trait aware. You can use it to create, modify or delete traits, you can work with its methods and also with classes that use them. But in the example we will use typed commands in order to explain its mechanism better.

Traits are declared with a syntax similar to a class declaration; for example:

```
Trait named: #TNewTrait uses: {} category: ''.
```

This piece of code would declare a new trait called TNewTrait. which doesn't use other traits and doesn't belong to any category. Once declared, the trait is placed at the system dictionary, like any other global variable. Then, we could add methods to the trait, as if we were adding methods to a normal class; we could do that in the Class Browser tool, or achieve the same result with a line like this:

```
TNewTrait addSelector: #aMessage withMethod:
aPreviouslyCompiledMethod.
```

Then, we would create a class that uses this trait:

```
Object subclass: NewClass uses: TNewTrait
instanceVariableNames: '' classVariableNames: ''
poolDictionaries: '' category: ''.
```

The difference with a normal class declaration is the “uses: TNewTrait” clause, that tells the system to create the class, but taking into account that this class implements the named trait.

When we send a message to an object, the search mechanism starts looking for the method in the method dictionary of the object's class. If it doesn't find it there, it would normally try to search for the method in the method dictionary of the parent class of the object's class. Instead of doing so immediately, the searching algorithm see if the object's class uses traits, and if so, it includes the trait composition in the search for the method. This searching through the trait composition could also lead to the recursion on other traits, that are used by the original trait. Then, and only after the search on the trait composition failed, it continues up to the parent class.

There is a penalty in the speed of the algorithm, as it has to search for a method in more places than it would otherwise have to. And there is also the overhead of calculating the trait composition as well. But this penalty only applies to classes that uses traits. Besides, as the search mechanism is cached, this penalty is mostly hidden behind the cache.

Going back to the example, if we tried to send the message: #aMessage to an instance of NewClass, the algorithm would start looking in its method dictionary and it would fail. Then, instead of going up to Object, it would check for the existence of #aMessage in TNewTrait, which would success, returning the address of the correct method to the caller. As can be seen from the example, although the implementation is different in Huemul than in Squeak, the result is exactly the same.

9 Graphical User Interface

The GUI (Graphical User Interface) defines the look and feel of an application. It is an important factor to the acceptance and usability of a system. In the times of the earlier Smalltalk implementations, the use of GUIs wasn't very widespread, and there weren't any standards. So, the first Smalltalk Systems implemented their GUI by their own. Nowadays, many Smalltalk implementations still use this approach. Huemul does not draw its graphics interface. It relies on the GTK [1] graphics toolkit to do the job. Huemul just uses a GTK wrapper ported from the original GTK Wrapper [8].

By using an existing GUI package, all of its features come for free into the hands of the Smalltalk programmer [14]. Many useful features like localization, theming, extensibility, etc. are standard. The user also benefits from this facility because they don't have to learn a different graphical platform. Huemul and all the developed applications look and feel as any other native application.

Huemul has a built-in set of development tools that cover the basic Smalltalk development experience. But they are built with the GTK framework. In this way, Huemul implements an intuitive user interface that is easy to adopt for the experienced user and the novice.

The GTK framework is built in C, but it specially designed to be wrapped by other languages. Most of the library functions take the first argument as the receiver (the Smalltalk self pseudo variable). Huemul maps Smalltalk classes to GTK classes. The hierarchy structure of the classes is rooted at the GPointer class. This class has a handler as its only variable. The handler is the pointer to the external GTK structure of the same name. All the GTK functions are wrapped by the external library call mechanism of Huemul. Implementing each of the GTK functions is straightforward thanks to this mechanism.

For example, the GTK window class is called: GtkWindow. In Huemul there is a class called GtkWindow that wraps around methods from its GTK counterpart. The function to resize the window is called gtk_window_resize. Its arguments are: the window itself, the width and the height. In Huemul the GtkWindow class has a #resize: method that accepts an instance of Point as its argument. This methods calls the GTK library function using self as the first argument, the point's x variable as the width, and the point's y variable as the height. All other methods are implemented in the same way.

Huemul is designed to be easy to connect to standard UNIX libraries. Huemul has a wrapper around the UNIX dynamic library loader. It also has a set of classes to access standard C structures and data types. The whole GTK wrapper is built around this concept, but there also other libraries like libc already wrapped in the system.
For example the GTK library is opened like this:

```
Gtk := ExternalLibrary new.

Gtk name: 'libgtk-x11-2.0.so'.

Gtk open.
```

And the function named above would be called with something like:

```
(Gtk functionNamed: 'gtk_window_resize') invokeWith:
windowHandler with: (ExternalAddress fromInteger:
```

```
aWidthInteger) with: (ExternalAddress fromInteger:
aHeightInteger).
```

Which is the Smalltalk equivalent of the following C code:

```
gtk_window_resize( windowHandler, aWidthInteger,
aHeightInteger );
```

Everything is done at runtime, no need to recompile anything at the VM level, or write an external plugin.

The GTK framework also supports the callback mechanism to send messages generated by GTK itself (and other sources as well). These callbacks are ready to be wrapped too. Huemul uses this mechanism to pass events from the GTK system to Smalltalk.

Almost the entire wrapper is written in Smalltalk. The only exception to this is the use of some macros that are defined in some GTK C headers. These macros are used to initialize some of the constants values used by GTK. If this macros were implemented in Smalltalk by the wrapper they would have been implemented hardcoded. This is not a good practice since the moment that something changes in the library, that would lead to a malfunction of the wrapper.

10 Portability

Huemul's minimalistic approach to the VM has the advantages already explained, but the main drawback of this approach is portability. The system is being developed in Linux with x86 CPUs of 32 bits. The CPU was chosen because it is in widespread use. The Operating System was chosen because of the facilities and tools available for the low level compiler environment. As the image contains natively compiled methods both the image and the VM are bound to the chosen platform.

Most of the Smalltalk implementations provide image compatibility between platforms, with or without a conversion mechanism. This is a fairly easy task for an interpreted image, but not for a natively compiled image. Huemul's high degree of integration with existing technologies produces a high dependency with the underlaying Operating System, making portability more difficult to achieve.

First we should analyze the portability of the VM, and then, the portability of the image, as both of them are platform dependent.

Huemul's VM is small and simple, and it is written in C by now. The elements bound to the platform in the VM are: the send mechanism, the use of setjmp/longjmp and the load of libraries at run-time. The send mechanism in Huemul is almost compatible with the C calling convention, but with some variants that should be considered but, as the C language has been ported without problems to many platforms, that would help us a lot. We are using gcc to compile the VM, and gcc is widely available in many platforms. The high degree of compatibility of the standard C libraries also helps with the setjmp/longjmp issue, as it is a standard C mechanism, there should not be a problem. The dynamic use of the libraries is something bound to the Operating System. All major Unix variants have some sort of mechanism of doing so. Mac OS X is Unix like, so there will be no problems. Windows has a similar mechanism. Other OS should be analyzed separately. So porting the VM should be as easy as considering a solution for each of these three issues. Once that is accomplished, the standard distribution and build mechanism of C sources of other applications apply.

The portability of the image is another matter. The main problem here is that the methods are saved with the image, and the code is compiled natively on each platform. One approach would be to populate the image with each platform's version of the method, and also include each of the platform's extensions in the compiler. With this mechanism the image could be distributed as is, throughout all the different platforms. This may be doable, but we don't consider it useful. Compiled methods take a lot of image space, with this approach, we would have to multiply the space requirement by the number of platforms supported. That is a lot of space wasted considering that we are going to use just one platform code set at a time. This issue may change if some sort of fragmentation of the image is applied. This would lead to a smaller footprint of the image, loaded at runtime.

A much more efficient approach would be to forget about image automatic portability, and provide some sort of conversion mechanism. This conversion mechanism could be though as a cross compiling procedure. If conversion time is an issue, the conversion mechanism could be focused to just some parts of the image, such as the compiler itself and the bootstrapping methods.

The embedded Smalltalk compiler is implemented in Smalltalk itself, and it uses Exupery as the machine code generator. Nowadays, Exupery just supports the generation of x86-32 machine code, but it is prepared to be extended to support other CPUs as well. If the target platform doesn't use a x86-32 CPU, we would have to extend Exupery to support it. Once this task is complete, we would just use Exupery as the cross compiler, that in conjunction with a tracer program, would help us to create the new image. This new image would be ready to be loaded in the new platform, but it would be incompatible with the actual one.

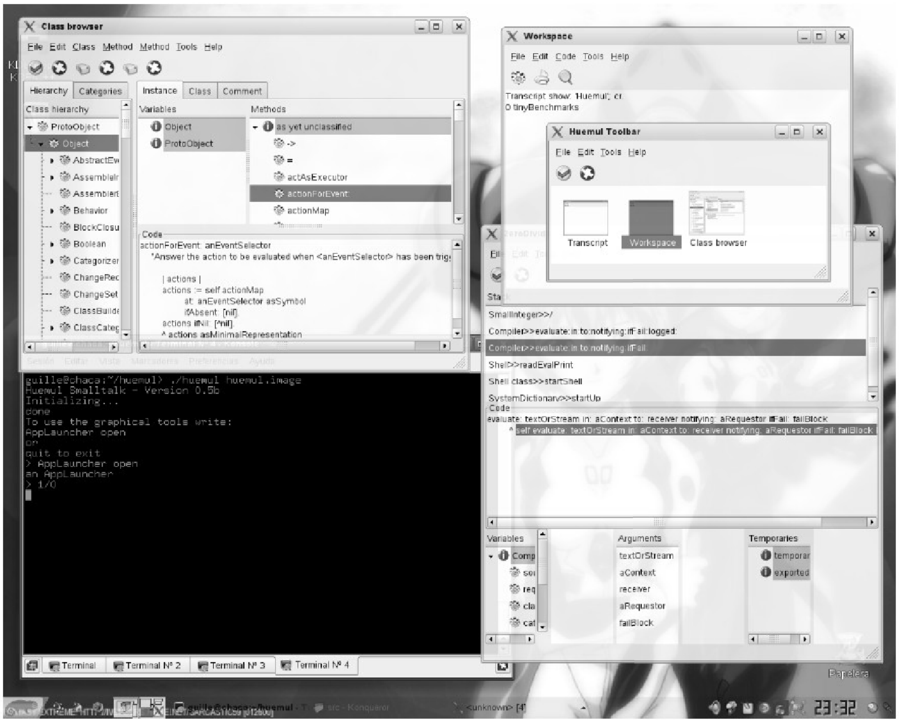
There are also Operating System compatibility details like GTK support. If we were porting to Windows, we would have to choose between a Unix-like environment like Cygwin [16] and use the already implemented wrapper or a new implementation of the native Win-32 API.

In summary, Huemul could be ported to new platforms. It depends on how much the new platform differs from the supported one. But with more or less difficulty the task could be achieved.

11 Current State of the System

Huemul is nowadays usable but incomplete. It still needs Squeak to create the bootstrapping image as Huemul's snapshot command sometimes fails. Development effort is focused in making Huemul standalone. This task includes to evolving the tools, and developing a good image format and a snapshot mechanism, designing a garbage collector, general platform stabilization, etc.

Huemul's home page is http://www.guillermomolina.com.ar/huemul. Huemul is downloadable either as sources by SVN or as a compressed archive that includes the image and the VM's binary. The latest archived version is 0.5b.



```
actionForEvent: anEventSelector
    "Answer the action to be evaluated when <anEventSelector> has been trig

    | actions |
    actions := self actionMap
        at: anEventSelector asSymbol
        ifAbsent: [nil].
    actions ifNil: [^nil].
    ^ actions asMinimalRepresentation
```

```
Transcript show: 'Huemul'; cr.
0 tinyBenchmarks
```

```
SmallInteger>>/
Compiler>>evaluate:in:to:notifying:ifFail:logged:
Compiler>>evaluate:in:to:notifying:ifFail:
Shell>>readEvalPrint
Shell class>>startShell
SystemDictionary>>startUp
```

```
evaluate: textOrStream in: aContext to: receiver notifying: aRequestor ifFail: failBlock
    ^ self evaluate: textOrStream in: aContext to: receiver notifying: aRequestor ifFail: failBlock
```

```
guille@chaca:~/huemul> ./huemul huemul.image
Huemul Smalltalk - Version 0.5b
Initializing...
done
To use the graphical tools write:
AppLauncher open
or
quit to exit
> AppLauncher open
an AppLauncher
> 1/0
```

Fig 6.1.

Huemul uses a standard MIT license [19], and reuses other developers' MIT licensed code. So it is free software, and it is available for general use.

Figure 6.1 shows a screenshot of Huemul in action. At the upper-left there can be seen the Class Browser, this tool is used to modify the classes and methods. To the right there is an Application Launcher. This tool is used as a starting point, to launch the other tools, and to control Huemul's execution. Behind it, there is a Workspace. The Workspace is a place to execute code to compute something useful, and develop bits of a program. Under the Workspace there is a Debugger window. This tool is the main debugging tool, whenever something goes wrong, this is the place to start. And to the left of the Debugger, there is the command line. The command line is the starting point of all other windows. Huemul can be used directly by entering commands there, without the use of the GUI.

12 Future Work

Many new technologies are still being considered. Future releases may include these technologies.

JIT. Currently there is no dynamic optimization in Huemul. The performance achieved could be enhanced if we apply some useful optimizations like incremental compiling, dynamic inlining, polymorphic inline caches, type feedback, etc. We could start by adding some optimizations already available with Exupery.

Garbage Collector. Huemul still lacks a good garbage collector (GC) . Different GC strategies are still being evaluated. A parallel garbage collector based on the "Mostly Parallel Garbage Collector" [12] seems to be the best choice, since this collector focuses on parallelization and scalability. And because there is a working plug-and-play library available from the authors of that work [7].

Image Format. Right now the image format is like a photograph of the running system. With this approach we are obligated to have an image loader in the VM. Our image loader has to apply relocation patches to the image before it can be used. We are considering adopting a standard executable format like elf [6] instead of our proprietary format. This would let us remove the loader from the VM, and make the Operating System do the job.

VM Compiler. Huemul has a tiny VM, but it still needs another language and compiler to develop it, as it is written in C. We would like to implement an extension to the actual compiler, that would override the standard send mechanism used for normal Smalltalk code. We would use such a compiler to port the VM to Smalltalk and embed it in the image. Leaving just an image bootstrapper in the executable. This mechanism would convert Huemul in a fully self-sustaining system.

13 Conclusions

This paper describes a new Smalltalk implementation focused on aggressive reusability. We analyzed the sources for functionality duplication of other implementations and proposed a new type of VM that is at the same time fast and small. This VM only implements the procedures that can not be pushed up into the image, and can not be pulled down to the Operating System and its libraries. The main difference with other implementation is the total lack of an interpreter. This layer is implemented by compiling Smalltalk code to native code without an intermediate compilation to bytecodes. The main drawback of this approach is the lack of automatic portability of the image, as it is bound to the host platform. There are also other parts of the VM that help reusability; like the use of native stack instead of Smalltalk contexts, the dynamic loading of libraries and functions, native threads, the GTK framework, etc. Overall, these techniques make Huemul integrate tightly with the platform and bring all the advantages that the platform provides to the Smalltalk world.

Acknowledgments

Thanks to Klaus D. Witzel and Sebastian Sastre for proof reading a draft version of this paper. Special thanks to Michael Haupt for reviewing the final version. And the European Smalltalk User Group (ESUG, http://www.esug.org) for supporting the presentation of this paper.

References

[1] The GTK Toolkit, `http://www.gtk.org/`
[2] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In: The OOPSLA 1997 Conference, Atlanta, Georgia (1997), `http://www.squeak.org/`
[3] Squeaks New Compiler, `http://smallwiki.unibe.ch/NewCompiler`
[4] Visual Works Smalltalk, `http://www.cincomsmalltalk.com/`
[5] Dolphin Smalltalk, `http://www.object-arts.com/`
[6] Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 (May 1995)
[7] A garbage collector for C and C++, `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`
[8] Squeak GTK Wrapper, `http://squeakgtk.pbwiki.com/`
[9] Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation (1983)
[10] Kampjes, B.: Exupery A Native Code Compiler for Squeak, `http://goran.krampe.se/exuperyDesign.pdf`
[11] Lienhard, A.: Bootstrapping Traits. M.S. thesis, University of Bern (2004)
[12] Boehm, H.-J., Demers, A.J., Shenker, S.: Mostly parallel garbage collectors (1991)
[13] Chambers, C., Ungar, D.: Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In: ACM SIGPLAN 1989 Conference on Programming Language Dessign and implementation, pp. 146–160 (June 1989)
[14] Babcock, M.: The Importance of the GUI in Cross Platform Development. Linux Journal 5(49) (May 1998)
[15] Smalltalk/MT, `http://www.objectconnect.com/`
[16] Cygwin, `http://www.cygwin.com/`
[17] Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming. In: A POSIX Standard for Better Multiprocessing (September 1996)
[18] Smith, D.E., Nair, R.: The Architecture of Virtual Machines. Computer 38(5), 32–38
[19] The MIT License, `http://www.opensource.org/licenses/mit-license.php`

SBCL: A Sanely-Bootstrappable Common Lisp

Christophe Rhodes

Department of Computing
Goldsmiths, University of London
New Cross, London SE14 6NW
`c.rhodes@gold.ac.uk`

Abstract. This paper describes the development of an implementation of Common Lisp with the peculiarity that it is bootstrappable neither solely from itself, nor from some other language, but rather from a variety of other Common Lisp implementations. We explain the motivation for this bootstrap strategy, discuss some of the technical details involved in achieving it, and attempt to assess the technical and social effects that it has had on the development of the implementation and on its user and developer community.

1 Introduction

The Lisp family of languages has a long history, being invented (or perhaps 'discovered') by John McCarthy in the late 1950s. Of the languages still in use today, only Fortran is older – though of course both modern Fortran and modern Lisp are worlds removed from the versions used in the 1950s. A wide range of dialects of Lisp were developed and disseminated in the following two decades, until in the 1980s moves towards consolidation between the various Lisp families occurred. The two most popular dialects of Lisp at the time of writing are Scheme and Common Lisp, both of which have international standards documents associated with them [1,2] (though in the case of Scheme there are also more lightweight community-led processes which have largely superseded the international standard [3,4]).

This paper is primarily concerned with implementation strategies for Common Lisp, and how those strategies affect the ease with which development of the implementations can occur. We do not address in this paper the implications of the details for other environments, presenting instead a case study of the social and technical effects observed in this single domain. In particular, we do not address the issues involved in cross-compiling Common Lisp for a different machine architecture, as these have been discussed elsewhere (see for example [5] and references therein).

Current Common Lisp implementations can usually support both image-oriented and source-oriented development. Image-oriented environments (for example, Squeak Smalltalk [6]) have as their interchange format an image file or memory dump containing all the objects present in the system, which can be later restarted on the same or distinct hardware. By contrast, a source-oriented environment uses individual, human-readable files for recording information for reconstructing the project under development; these files are processed by the environment to convert their contents into material which can be executed.

It would be unusual to find a Common Lisp application programmer these days working in an image-oriented manner; it is far more usual to work with source code stored in files and loaded using `compile-file`, than to define functions exclusively using the evaluator or read-eval-print loop and to store state by saving memory images, though the functionality of saving images is retained in contemporary Common Lisp implementations (despite not being part of standardized functionality) and is most often used as a deployment strategy.

Of course, Common Lisp application programmers are used to making incremental modifications to their software; Lisp environments are renowned for having the facilities to develop functions one at a time, coupled with the ability to use the image's introspective capabilities for finding information about callers and callees of functions and where variables are bound, for providing views of data structures (through an inspector or through more specialized browsers for classes, generic functions and the like), as well as for rapid recompilation and incorporation of modifications.

However, as image formats are not standardized, and indeed historically do change between releases of Common Lisp implementations, the application programmer is used to verifying from time to time that their current sources compile cleanly from scratch – that is, that no dependency on something which is only present in the image has been introduced in the sources.[1]

In the sphere of Lisp implementations themselves, however, this picture is reversed: it is somewhat unusual to find a Lisp implementation, written primarily in Lisp, which does not have a flavour of this image-oriented development within it. The typical build process in this case involves using a host lisp of the same implementation (but an earlier version), then mutating it incrementally to the point where it matches the new sources sufficiently to be able to compile those new sources, and then dumping an image. The mutation is in general different for each particular change at the source code level – many changes require no mutation at all, while changes to compiler-internal data structures may require very involved mutations: we give an example in Section 4.1.

This paper discusses Steel Bank Common Lisp (SBCL), a Common Lisp implementation which is largely written in Lisp, while limiting and containing the image-based incremental modification of its own self as part of its build process to a strictly manageable level: the outcome of the build does not depend on the state of the host lisp compiler. The rest of this paper is organized as follows: in Section 2, we describe the history and current state of Steel Bank Common

[1] A simple but real-world example of this comes from the abstraction of a syntactic pattern into a macro which has uses before its definition, because the most expedient place to put that definition was not in the first source file to be compiled from scratch. SBCL has a number of source files with prefix `early-` (for example, `early-package.lisp` and `early-setf.lisp`) for the purpose of holding definitions which must be seen early in the build.

Lisp; then in Section 3, we go into the detail of how SBCL is built, comparing our approach with other Common Lisps. We discuss the benefits and drawbacks of this build process in Section 4, and draw conclusions in Section 5.

2 Steel Bank Common Lisp

Although Steel Bank Common Lisp (SBCL) is a relatively new Common Lisp implementation, it shares much code and a long development history with its closest relative, Carnegie-Mellon University Common Lisp [7] (CMUCL). CMUCL was a project funded by DARPA under CMU's "Research on Parallel Computing" contract, and began life as SPICE Lisp. Under that contract, CMUCL was developed continuously at Carnegie-Mellon University from the early 1980s until funding was stopped in 1994; at that point, CMUCL support at CMU was discontinued, but the project continues to this day, with a group of users and developers collaborating over the Internet.

SBCL was announced as a CMUCL variant with a 'clean' bootstrap process, in December 1999 by Bill Newman on the CMUCL developers' mailing list [8]. Since then, it has been developed further, initially by Newman alone, then with an increasing number of contributions from individuals, starting from the move to public CVS hosting on SourceForge in September 2000. The number of contributors has since risen significantly; at the time of writing, there are 23 people with commit privileges to the master CVS repository, while over the course of 2007 code contributions from over 40 people were incorporated.

The system as of early 2008 contains approximately

- 90,000 lines of lisp code implementing the 'standard library', excluding the Common Lisp Object System (CLOS);
- 60,000 lines of lisp code implementing the compiler (and related subsystems, such as the debugger internals);
- between 10,000 and 20,000 lines of lisp code per architecture backend implementing the code generators and low-level assembly routines;
- 20,000 lines of lisp code implementing CLOS;
- 20,000 lines of lisp code implementing contributed modules or 'extras';
- 35,000 lines of C and assembly code, for services such as signal handling and garbage collection;
- 30,000 lines of shell and lisp code for regression tests.

It is perhaps worth discussing briefly why there is a substantial component written in C and assembler: some 10% of the total. Partly this is because of the large number of architecture/operating system pairs supported; each such pair contributes some 200 lines of code implementing platform-specific operators (such as finding the faulting address from within a memory fault handler function); additionally, each supported operating system (of which there are five) contributes 2000 lines, and each architecture (of seven) 2500 lines. The Garbage Collector is about 8000 lines of code, and is written in C for essentially pragmatic reasons: when the system is unstable enough for the GC to require debugging, using an external debugger (such as the GNU debugger, `gdb`) removes some uncertainty in the debugging process: and such external debuggers are better tailored to debugging C than Lisp.

We discuss the technical details of SBCL's build process in more detail in the next section; to give a high-level overview, SBCL's build achieves independence from the host lisp used to build it by embedding an SBCL compiler within the host, before using that embedded compiler to generate a fresh, standalone SBCL image. These two compilers, embedded and standalone, are generated from the same source code files; this works because we have effectively done the same as is commonly described as idiomatic Common Lisp programming style: to write a domain-specific language for solving one's problem, then solving the problem in that language – but in our case, the domain-specific language happens to be Common Lisp itself.

3 The SBCL Build Process

3.1 Build Processes of Other Lisps

We discuss the build processes and implementation strategies of other contemporary Common Lisps, in order to put SBCL's strategy in context. For more general information about these Lisps, see a recent survey of implementors conducted in late 2007 [9].

We can briefly summarize the implementation strategies of current Common Lisp implementations by dividing them into two categories: those which have significant portions implemented in languages other than Lisp, and those which are primarily Lisp-based. (The key to the division is whether there is enough implemented in the other language to implement an interpreter, or whether all Lisp evaluation is written in Lisp).

- Other-language based:
  - C implementation, C compiler: GCL, ECL (Kyoto CL derivatives)
  - C implementation, bytecode compiler: GNU CLISP
  - Java implementation, Java bytecode compiler: ABCL
  - C++ implementation, native compiler: xcl
- Implemented primarily in Lisp:
  - only buildable in themselves, using image-based techniques: Allegro Common Lisp[2], LispWorks[2], CMUCL, Scieneer CL[2], Clozure CL;
  - buildable in several Common Lisps: SBCL.

For the implementations where there is an evaluator in a non-Lisp language, the bootstrapping strategy is straightforward: building enough of an environment in that other language to be able to evaluate Lisp, and then build up the rest through successive evaluation. Note that this build strategy does not necessarily

[2] The closed-source nature of these implementations prevents the author from making any authoritative statement, but anecdotal evidence suggests that placing these implementations in this category is correct.

involve very much code in the 'other' language (see for example Lisp500[3], which has 500 lines of highly-obfuscated C); however, for systems intended for real-world usage, the figure is significantly higher: GNU clisp has 180,000 lines of 'D' – which is then preprocessed into C; gcl has 75,000 lines of C code implementing the compiler core, along with another 1,000,000 lines of C code from libraries for binary creation and accurate multiprecision arithmetic: binutils and gmp.

Part of our motivation for working on SBCL rather than other implementations is that we believe that Lisp is a good language for general programming, including the writing of interpreters and compilers and for manipulating complex data structures, and that therefore it would be a shame not to use it to the full in the development of a Lisp implementation: but the determinism granted by SBCL's build procedure as described in this paper is also important.

3.2 Building SBCL Itself

The SBCL build process is not greatly dissimilar to the build process of compilers such as `gcc` [10, Chapter 11]; there is an extra dimension to it, however, thanks to the accumulation of state during a compilation and loading process, which is not present in static C-like languages: this is the underlying reason for having namespaces beginning `sb!` in the cross-compiler (corresponding to `sb-` in a running SBCL); we discuss this further in Section 3.3. The following diagrams illustrate the build process; in them, files either produced or used in the process are represented as ellipses, while Lisp processes themselves are rectangles.

The first step (Figure 1) involves using the host compiler to compile and load SBCL's source files, which produces a cross-compiler (denoted by `xc`) running as an application inside the host. This step also builds other applications, including `sb!fasl:genesis`, which will be used later. It is then possible to introspect over the definitions of the data structures for the target lisp, and produce a set of C header files describing Lisp data structure layouts.

These C header files, along with the C source and assembly files, are then used (Figure 2) to produce the `sbcl` executable itself. The executable is as yet not useful; while it provides an interface to the operating system services, and a garbage collector, it requires a compatible Lisp memory image (produced in the next steps) to function. Additionally, a small C program is compiled and executed, generating Lisp source code describing system constants and types.

Next, the cross-compiler version of `compile-file` is used to compile the SBCL sources again, along with the generated Lisp source file from the previous stage (Figure 3). This produces a set of object files ('FASL files' in Lisp terminology, here given the filesystem extension `.lisp-obj`). Although in Figure 3 the host and cross-compiler are displayed as though unchanged, in fact there are some fine differences between the cross-compiler at the start and the end of this process: the cross-compiler will have acquired new constant definitions (from the generated

[3] Available at `http://www.modeemi.fi/~chery/lisp500/`. Lisp500 is not intended to be more than a 'toy' Common Lisp; its implementation is such that `eval` is written in C.

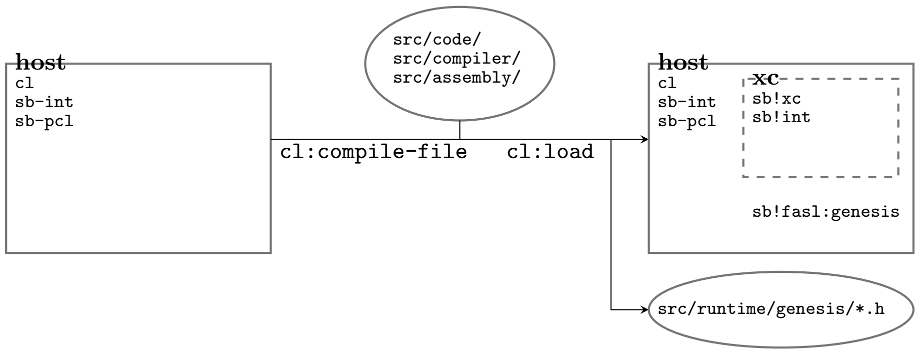


Fig. 1. The *host-1* build stage

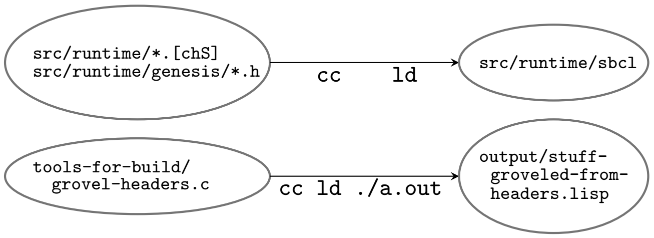


Fig. 2. The *target-1* build stage

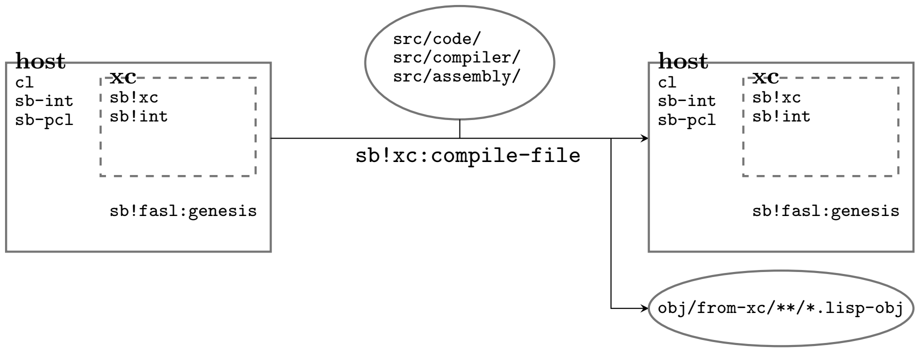


Fig. 3. The *host-2* build stage

lisp file, for example). However, the functions and data structure definitions on the host, including those of the cross-compiler application, are unchanged by this process.

The next stage (Figure 4) simulates the act of loading the FASL files and saving a memory image, using the `genesis` application. We cannot simply load those FASL files using the standard Common Lisp `load` function, because `load`

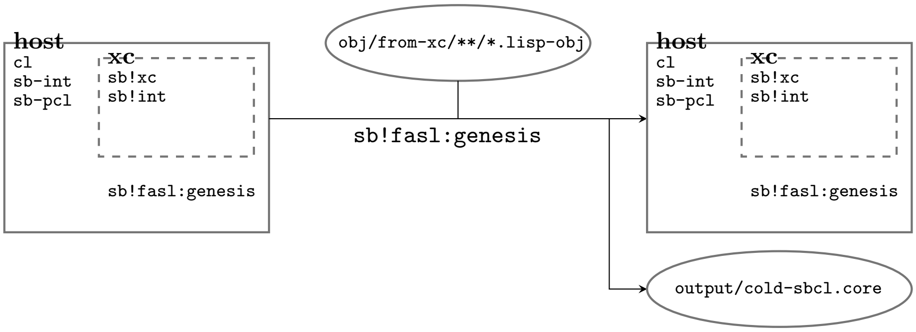


Fig. 4. The *genesis-2* build stage

uses the host compiler's FASL file format, not SBCL's format. Nor can we use the `load` function of the target image, because that target image does not yet exist: indeed, its creation is the purpose for wanting to load these FASL files in the first place.

Instead, we effectively build up the memory image by pseudo-loading each FASL file in the specified build order: we represent the memory areas of the target lisp as vectors of bytes, and perform the actions that would occur on loading the FASL file, not on the host lisp's data structures but rather on the representation of the appropriate memory area. There are actions that cannot be simulated in this way, such as function calls to arbitrary target functions – these actions are deferred until the initial function of the target image is run – but in particular function definition can be performed, so that the initial function is capable of calling other, named functions.

Once all of the FASL files have been processed in this way, the memory areas are saved to file in the format expected by the `sbcl` binary created earlier, producing a 'cold core'; there is a special case for dumping symbols of the `sb!xc` package, which are dumped as though they were in the `cl` package.

The `sbcl` binary is run with the 'cold core' as its memory image: this core has a particular entry point or 'toplevel', which performs a sequence of actions to allow the processing of Common Lisp code. For instance, in the `genesis` sequence, no top-level forms from the FASL files have actually been run, because there is no way that they can be run using only the host lisp's facilities. Instead, they are deferred to this time. There are many other similar kinds of fixups which need to be performed at this time, and in a particular order; this 'cold init' phase is probably the most fragile portion of the SBCL build currently, and it is also the hardest to debug (because if it goes wrong, there will be no helpful Lisp debugger).

Finally, after the 'cold init' phase, the packages can be renamed to their final names (so that `sb!` prefixes are converted to `sb-`), and the specialized version of Portable Common Loops (PCL) for SBCL compiled and loaded; finally, a new memory image is saved as `output/sbcl.core`, and the build process is complete (Figure 5).

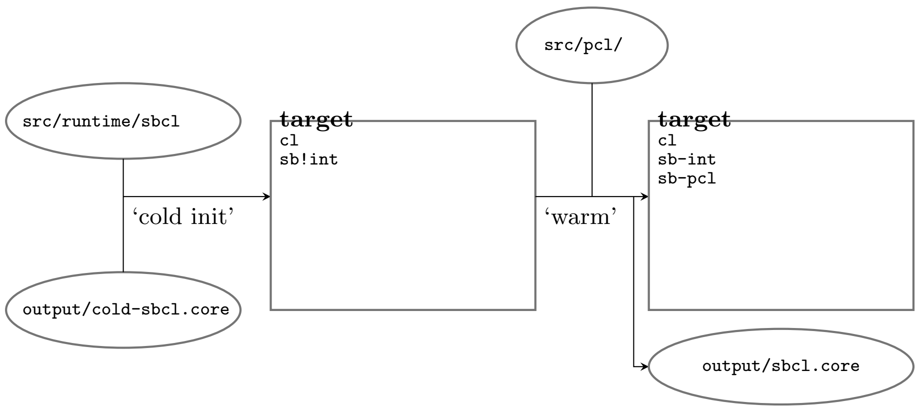


Fig. 5. The *target-2* build stage

3.3 Separation of Host and Target

During the SBCL build process, there needs to be a clear separation of the host and target worlds. In particular, if the SBCL build host is another version of SBCL, the build must neither use any properties of the host SBCL, nor mutate any of its structures.

To a large extent, this is achieved through one simple mechanism: packages which in a fully-built SBCL have prefix `sb-`, during the build have names beginning `sb!` – so that, for example, the package named `sb!kernel` during the build corresponds to the `sb-kernel` package in a running SBCL. This package name transformation occurs very early during the initial 'cold' boot of the lisp image dumped by the genesis phase of the build process.

The handling of the `*features*` variable is conceptually similar. The standard facility for conditional compilation in Common Lisp (really conditional code inclusion, somewhat like the C preprocessor's `#ifdef` construct) dispatches on the host's value of `*features*`, including or commenting out code based on feature expressions following `#+` and `#-` reader macros. The SBCL build provides analogous `#!+` and `#!-` reader macros for conditional reading of forms based on the target value of `*features*`, so that (for example) the cross-compiler can include architecture-specific optimizations or operating system specific system calls.

There is a complication, however, for handling routines that will be part of the `common-lisp` package in the target lisp: we clearly cannot overwrite any of the host compiler's functions or constants, but we need to be able to refer to target versions of these: for instance, so that the cross-compiler can compile `defmacro` forms. The solution there is to have a shadow `common-lisp` package named `sb!xc`, which only exists during the build process: operators which are needed for the build but which collide with `common-lisp` operators in the host lisp are placed there, and genesis has a special case for dumping symbols from the `sb!xc` package, so that the cold boot phase runs with all the necessary `common-lisp` operators already present.

To give a concrete example, the operator `cl:defconstant`, when evaluated by the host compiler, will define a constant in the host's world. The operator `sb!xc:defconstant`, when evaluated by the cross-compiler, will define a constant in the cross-compiler; further, when a call to `sb!xc:defconstant` is *compiled* by the cross-compiler, it will be as if `cl:defconstant` has been run, once the compiled call has been 'evaluated' during the genesis and cold boot phase.

As well as those simple special cases of host and target separation, there are a couple of slightly more complicated cases that nevertheless need some care. One is `eval-when`, which can cause confusion in even seasoned Common Lisp programmers. The `eval-when` operator allows the programmer to specify that certain forms should be executed when the file containing the form is file-compiled (`:compile-toplevel`) or when the resulting FASL file is loaded (`:load-toplevel`), or both – as well as when in a normal execution context (`:execute`, which is never active in the SBCL build process itself). In the context of phase separation and maintainability, the `eval-when` operator may not be the best solution [11], but it is available in conforming Common Lisps, so the SBCL build process can rely on it.

Consider the following example fragment:

```
(eval-when (:compile-toplevel :load-toplevel)
  (defun foo (x) (+ 7 x)))
```

Since the cross compiler cannot run any code of its own, for the `:compile-toplevel` case it must use `cl:defun` (*i.e.* the host's `defun`) here, not `sb!xc:defun`. However, for the `:load-toplevel` case, we are compiling a call that would eventually be executed by the target, so the cross-compiler's version of the macroexpander for `defun` must be used.

The other case revolves around `make-load-form`. The build process makes use of `make-load-form` extensively, for dumping compiler structures into FASL files. However, care must be taken here, because the host compiler's implementation of `make-load-form-saving-slots` is not necessarily compatible with the cross-compiler's – and yet the same code must be capable of dumping both compatibly for the host, while building the cross-compiler, and compatibly with the target, while building the target compiler. This is achieved by having a `make-load-form` method which dispatches on the presence of the `:sb-xc-host` on the *host's* lisp's `*features*` variable, which indicates which phase in the build is currently being executed.

3.4 Lisp Library Differences

There are other non-portabilities in the SBCL build process that are addressed at the time of writing to a greater or lesser extent.

Constant-folding and, potentially, type derivation in the cross-compiler will interfere with correct operation if the host Lisp's model of `float` subtypes is not the same as the target's. SBCL divides the `float` type into two subtypes: `single-float` (the same as `short-float`) for IEEE single floats, and `double-float` (the same as `long-float`) for IEEE doubles. Compiling from a host lisp which had different interpretations of `single-float` and `double-float` to SBCL's would be challenging, though note that SBCL's code is not sensitive to the host lisp's interpretation of `short-float` and `long-float`.

One observation remains: it is surprisingly difficult to write theoretically portable code for handling a large block of memory. Common Lisp provides the abstraction of a `vector`, of course, but the standard only specifies that the maximum size of a vector offered by the implementation must be 1024 elements or greater. While the author has not encountered a system where that limit is quite so low, implementations with an `array-total-size-limit` of the order of 2^{24} (on a 32-bit implementation where vectors are represented in memory with a header word consisting of an 8-bit type tag and 24 bits for the vector length) prompted a rewrite of the genesis phase to use 'big vectors', so that an in-memory data structure representing the bytes of the target lisp image (typically 20MB in size) could be built without falling foul of the array limit. The current 'big vector' implementation, representing linearly-addressible space as a vector of vectors of bytes, is in principle not sufficient, as the maximum space portably representable with such a data structure is 1MB (though in practice no implementation has an `array-total-size-limit` as low as 1024, and so our vector-of-vectors implementation is sufficient).

4 Discussion

4.1 Advantages

The immediate benefit of a straightforwardly reproducible build process is that no-one need learn the intricacies of the build process to contribute small, non-invasive patches. To a large extent, the author believes that this single fact is responsible for the current relative popularity of SBCL among Common Lisp implementations, and perhaps has even contributed to the increase in interest of Common Lisp as a whole.

Additionally, the straightforward build process, and in particular the clear separation between the build host and the target, allows for a smoother path for more invasive patches. As a simple example, there is no difficulty at all in renumbering the tags for type information, which is useful to allow more efficient assembly sequences for type checking.

In several cases the clear separation has resulted in bug fixes that were both cleaner and more straightforward to develop when compared to the more image-based Lisp implementations. For instance, there was a bug in both SBCL and its parent CMUCL in the handling of accessors for structures when name clashes occur through inheritance: a corner case to be sure, but one that was detected and fixed in December 2002 (SBCL) and January 2003 (CMUCL), by adding a slot to record inherited accessor information to the structure representing descriptions of defstructs.

It is in dealing with this kind of circularity that the simplicity of SBCL's build truly wins. In the case of SBCL, the fix was simple to implement: the slot was added to the sources, and then the sources were recompiled using the standard build procedure. In the case of CMUCL, however, in addition to changing the sources, two 'bootfiles' were necessary, and the system needed to be built at least twice: once loading the first bootfile beforehand (and interactively choosing a particular restart from an error condition); and once loading the second. The fix for the bug, which was very simple conceptually, was developed for SBCL by someone not in the development team; for CMUCL, it required an expert in the CMUCL build process itself to implement the bootfiles, demonstrating the principle that there can be a large impedance to contributions from newcomers.

Empirically, we can also say that SBCL as a whole, including its approach towards buildability, supports a community of users and developers which spans the gamut between experimentation with language and environment (examples include modular arithmetic, sequences [12], generic specializers [13]) and industrial use (as in ITA Software's QPX and RES products [9]). It should of course be noted that we cannot point to cause and effect here: there were all sorts of other factors allowing the SBCL (and Common Lisp) community to grow in number and scope, notably the development of a test suite for standardized functionality [14] and success stories from other Lisp vendors (see references in [9] for information current as of late 2007).

4.2 Downsides

The most obvious downside to the reworking of the build process described here is that each build inherently takes twice as long as in simple image-based systems: the compiler must effectively be built twice. These days, thanks to ubiquitous fast processors, this problem is much less evident than when the project started in 1999; the author remembers typical builds of upwards of an hour for SBCL, where its close relative CMUCL took on the order of 10 minutes (there were other factors for the more-than-doubling, including peak higher memory usage). Over the years, as processors have become faster, available memory has become greater, and SBCL's compiler has been optimized, there is only a small difference in build time[4]: and of course there is no need to think about *how* to build SBCL, or whether some kind of bootstrap script is needed – it can simply be set off.

Although a working model of SBCL is built as the cross-compiler, this is an incomplete model, and in particular it does not include CLOS (but it does include some hand-rolled object systems, in particular for implementing `subtypep`). SBCL's implementation of CLOS is derived from the Portable Common Loops

[4] On an Intel Pentium-M with clock speed 1.7GHz, building SBCL 1.0.16 using SBCL 1.0.15 took 541 seconds of user time, while building CMUCL 19d_p2 (without any extras such as CLX, the Motif bindings or Hemlock) with CMUCL 19d took 485 seconds of user time. Note that the CMUCL build involves three compilation phases: given sources corresponding sufficiently closer to the CMUCL compilation host, some of those phases may be unnecessary – but the default is to compile three times.

[15] package, with many modifications to improve conformance with the description of the Metaobject Protocol for CLOS [16]. In particular, the implementation of CLOS has inherent metacircularities, and representing this metacircularity in a host-independent way has not yet been addressed. Allowing CLOS to be part of the cross-compiler would probably simplify some portions of the logic within SBCL's compiler and type inferencer, at the possible cost of making the bootstrapping procedure rather more complex.

5 Conclusions and Further Work

We believe that Common Lisp itself is well suited to the domain of Common Lisp compilers, and as such it is an appropriate technique to use Common Lisp as the implementation language for a Common Lisp implementation. We have further shown that it is possible to avoid a circular dependency: SBCL is written in Common Lisp, not in its own dialect.

We have presented evidence that this has some positive effects; the removal of the cognitive overhead in working out whether any given change to the system requires special measures to build allows both for more rapid development by individuals and for an easier path for newcomers to get involved with the system – a particularly critical requirement given the relative lack of popularity of Common Lisp these days, and the fact that the Common Lisp implementation market is fragmented, with numerous proprietary and open implementations competing for market share. Though no single instance of SBCL is self-sustaining, the system consisting of the SBCL software, its users and its developers has an improved self-sustainability thanks to the conceptual simplicity of the maintainable modification of the software itself.

One straightforward improvement to the self-sustainability of the system would be a compilation mode for SBCL which would remove all non-deterministic elements from the FASL files produced by the cross-compiler (for example, the corresponding source code pathnames, or a build timestamp); this would allow for more straightforward testing of Common Lisp implementations which are currently not capable of building the system, and determining whether the problem lies in those implementations or an as-yet unidentified unportability in SBCL itself.

Acknowledgments

Thanks are due first and foremost to Bill Newman, who began the process of producing Steel Bank Common Lisp in 1999, and who encouraged and restricted further development with impeccable taste. The many other collaborators on that project, too numerous to name here, have provided a superb community within which to do interesting and meaningful work. We also thank the reviewers for their helpful suggestions. The author acknowledges the support provided by a PPARC studentship and EPSRC research projects (grants GR/S84750/01 and EP/E02274X/1), which have allowed the indulgence of an enjoyable hobby.

References

1. IEEE: IEEE Standard for the Scheme Programming Language. Technical Report, pp. 1178–1990. IEEE, Los Alamitos (1990)
2. Pitman, K., Chapman, K. (eds.): Information Technology – Programming Language – Common Lisp. Number 226–1994 in INCITS. ANSI (1994)
3. Kelsey, R., Clinger, W., Rees, J.: Revised^5 Report on the Algorithmic Language Scheme. Higher-Order and Symbolic Computation 11(1) (1998)
4. Sperber, M., Dybvig, R.K., Flatt, M., van Straaten, A.: Revised^6 Report on the Algorithmic Language Scheme. Technical report (2007), `http://r6rs.org`
5. Brooks, R.A., Posner, D.B., McDonald, J.L., White, J.L., Benson, E., Gabriel, R.P.: Design of an Optimizing, Dynamically Retargetable Compiler for Common Lisp. In: Lisp and Functional Programming, pp. 67–85 (1986)
6. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. ACM SIGPLAN Notices 32(10), 318–326 (1997)
7. MacLachlan, R.: CMUCL User's Manual. Technical Report CMU-CS-92-161, Carnegie-Mellon University (1992), `http://common-lisp.net/project/cmucl/doc/cmu-user/`
8. Newman, W.H.: It's alive! (SBCL, a CMUCL variant which bootstraps cleanly). Message-ID: 19991214185346.A1703@magic.localdomain on cmucl-imp@cons.org (December 1999)
9. Weinreb, D.L.: Common Lisp Implementations: A Survey (2007), `http://common-lisp.net/~dlw/LispSurvey.html`
10. von Hagen, W.: The Definitive Guide to GCC. Apress (2006)
11. Flatt, M.: Composable and Compilable Macros: You Want It When? In: International Conference on Functional Programming, pp. 72–83 (2002)
12. Rhodes, C.: User-extensible Sequences in Common Lisp. In: International Lisp Conference Proceedings (2007)
13. Newton, J., Rhodes, C.: Custom Specializers in Object-Oriented Lisp. In: European Lisp Symposium Proceedings (2008)
14. Dietz, P.: The GNU ANSI Common Lisp Test Suite. In: International Lisp Conference Proceedings (2005)
15. Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F.: Common Loops: Merging Lisp and Object-Oriented Programming. In: OOPSLA 1986 Proceedings, pp. 17–29 (1986)
16. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge (1991)

Reflection for the Masses

Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt

Vrije Universiteit Brussel, Programming Technology Lab, B-1050 Brussel, Belgium
{charlotte.herzeel,pascal.costanza,tjdhondt}@vub.ac.be

Abstract. A reflective programming language provides means to render explicit what is typically abstracted away in its language constructs in an on-demand style. In the early 1980's, Brian Smith introduced a general recipe for building reflective programming languages with the notion of *procedural reflection*. It is an excellent framework for understanding and comparing various metaprogramming and reflective approaches, including macro programming, first-class environments, first-class continuations, metaobject protocols, aspect-oriented programming, and so on. Unfortunately, the existing literature of Brian Smith's original account of procedural reflection is hard to understand: It is based on terminology derived from philosophy rather than computer science, and takes concepts for granted that are hard to reconstruct without intimate knowledge of historical Lisp dialects from the 1960's and 1970's. We attempt to untangle Smith's original account of procedural reflection and make it accessible to a new and wider audience. On the other hand, we then use its terminological framework to analyze other metaprogramming and reflective approaches, especially those that came afterwards.

1 Introduction

Programming languages make programming easier because they provide an *abstract* model of computers. For example, a Lisp or Smalltalk programmer does not think of computers in terms of clock cycles or transistors, but in terms of a virtual machine that understands s-expressions, and performs evaluation and function application, or understands class hierarchies, and performs message sending and dynamic dispatch. The implementation of the particular programming language then addresses the actual hardware: It is the interpreter or compiler that translates the language to the machine level.

Programming languages do not only differ in the programming models they provide (functional, object-oriented, logic-based, multi-paradigm, and so on), but also in fine-grained design choices and general implementation strategies. These differences involve abstractions that are implemented as explicit language constructs, but also "hidden" concepts that completely abstract away from certain implementation details. For example, a language may or may not abstract away memory management through automatic garbage collection, may or may not support recursion, may or may not abstract away variable lookup through lexical scoping, and so on. The implementation details of features like garbage collection, recursion and lexical scoping are not explicitly mentioned in programms and are thus said to be *absorbed* by the language [1]. Some languages absorb less and reveal more details about the internal workings of their implementation (the interpreter, the compiler or ultimately the hardware) than others.

We know that all computational problems can be expressed in *any* Turing-complete language, but absorption has consequences with regard to the way we think about and express solutions for computational problems. While some kinds of absorption are generally considered to have strong advantages, it is also obvious that some hard problems are easier to solve when one does have control over the implementation details of a language. For example, declaring *weak references* tells the otherwise invisible garbage collector to treat certain objects specially, using the *cut* operator instructs Prolog to skip choices while otherwise silently backtracking, and *first-class continuations* enable manipulating the otherwise implicit control flow in Scheme. When designing a programming language, choosing which parts of the implementation model are or are not absorbed is about finding the right balance between *generality* and *conciseness* [2], and it is hard to determine what is a good balance in the general case.

A reflective programming language provides means to render explicit what is being absorbed in an *on-demand* style. To support this, it is equipped with a model of its own implementation, and with constructs for explicitly manipulating that implementation. This allows the programmer to change the very model of the programming language from within itself! In a way, reflection strips away a layer of abstraction, bringing the programmer one step closer to the actual machine. However, there is no easy escape from the initially chosen programming model and its general implementation strategy: For example, it is hard to turn an object-oriented language into a logic language. Rather think of the programmer being able to change the fine-grained details. For example, one can define versions of an object-oriented language with single or multiple inheritance, with single or multiple dispatch, with or without specific scoping rules, and so on. In the literature, it is said that a reflective language is an entire region in the *design space of languages* rather than a single, fixed language [3].

In order to support *reflective programming*, the implementation of the programming language needs to provide a *reflective architecture*. In the beginning of the 1980's, Smith et al. introduced *procedural reflection*, which is such a reflective architecture that introduces the essential concepts for building reflective programming languages. Since the introduction of procedural reflection, many people have used, refined and extended these concepts for building their own reflective programming languages. As such, understanding procedural reflection is essential for understanding and comparing various metaprogramming and reflective approaches, including macro programming, first-class environments, first-class continuations, metaobject protocols, aspect-oriented programming, and so on. Unfortunately, the existing literature of Smith's original account of procedural reflection is hard to understand: It is based on terminology derived from philosophy rather than computer science, and takes concepts known in the Lisp community in the 1960's and 1970's for granted that are hard to reconstruct without intimate knowledge of historical Lisp dialects.

In this paper, we report on our own attempt to untangle and reconstruct the original account of procedural reflection. Our approach was to actually reimplement a new interpreter for 3-Lisp, using Common Lisp as a modern implementation language. In our work, we also build upon experiences and insights that came after the introduction of procedural reflection in [1], and also add some refinements based on our own insights. Additionally we use the concepts introduced by procedural reflection to analyze various other metaprogramming and reflective approaches, especially those that came after Smith's conception of procedural reflection.

2 Self Representation for a Programming Language

By introducing *procedural reflection*, Smith introduced a general framework for adding reflective capabilities to programming languages. In order to turn a programming language into a reflective version, one must extend the language with a model of the same language's implementation. That self representation must be *causally connected* to the language's implementation: When we manipulate the self representation, this should be translated into a manipulation of the *real* implementation. What parts of the implementation are possibly relevant to include in the model can by large be derived from mapping the non-reflective version of the programming language to Smith's theory of computation.

2.1 A Model of Computation

According to Smith [1], computation can be modeled as a relational mapping between three distinct domains: a *syntactic domain*, an internal representational domain (the *structural field*) and the "*real world*". The syntactic domain consists of program source code. The structural field consists of all runtime entities that make up the language implementation. In extremis, this includes the electrons

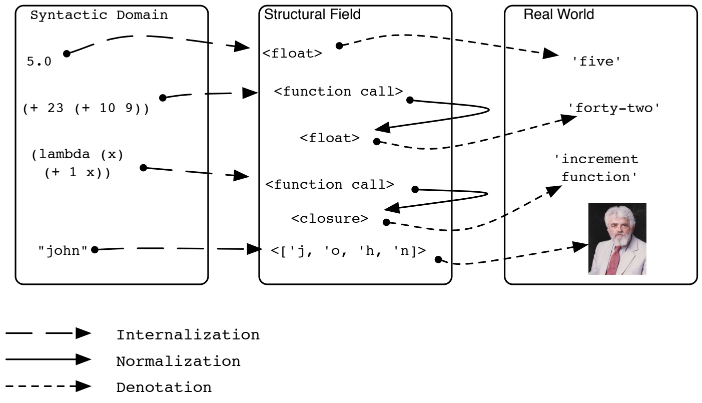


Fig. 1. Smith's theory of computation

and molecules of the hardware on which the interpreter runs. However, for the purpose of procedural reflection, the structural field typically consists of high-level data structures for implementing the values native to the programming language, such as numbers, characters, functions, sequences, strings, and so on. The real world consists of natural objects and abstract concepts that programmers and users want to refer to in programs, but that are typically out of reach of a computer.

Fig. 1 shows a graphical representation for computation as a mapping from programs, to their internal representation, to their meaning in the real world. The three labelled boxes represent each of the domains. The collection of arrows depict the mapping. The figure is meant to represent a model for computation rendered by a Lisp interpreter, written in C – hence the s-expressions as example elements of the syntactic domain. The `<type>` notation is used to denote an instance of a particular C type or struct. For example, the "number 5" is written as "5.0" in the programming language. Internally, it is represented by a `C` value of type `float` and it means, as to be expected, "number 5".

Each mapping between two domains has a different name. Mapping a program to its internal representation is called *internalization*, also typically called *parsing*. The mapping from one element in the structural field to another one in the structural field is called *normalization*. Normalization plays the role of *evaluation* and reduces an element in the structural field to its "simplest" form. For example, the s-expression `(+ 23 (+ 10 9))` is internalized to some structure representing a "procedure call". When that element is normalized, it is mapped to 42 – or better: To the internal representation of "42". The mapping from an internal representation to its meaning in the world is called *denotation*. In general, we cannot implement an interpreter that performs denotation. This is rather something only human beings can do. What real implementations do instead is to mimic denotation, which is called *externalization* [4]. For example, Common Lisp is built around the concept of "functions", and Common Lisp programmers do not want to be bothered by the internal implementation of functions as instances of a C struct for closures. Thus, when a function is printed, something like "<function:f>" is displayed. By limiting the kinds of operations available in Common Lisp for manipulating closure values and making sure they are printed in a way that does not reveal anything about their implementation, Common Lisp provides programmers the illusion of "real" functions. Such absorption and the provision of an "abstract" model of the hardware is the entire purpose of programming languages. By adding reflection to the language, we (purposefully) break that illusion.

2.2 Reflection

Reflection allows programmers to program as if at the level of the language's implementation. Smith distinguishes between two kinds of reflection. *Structural reflection* is about reasoning and programming over the elements in the internal domain, i.e. about inspecting and changing the internal representation of a program. On the other hand, *procedural reflection*, later also called *behavioral* *reflection*, is concerned with reasoning about the normalization of programs. The former allows treating programs as regular data and as such enables them to make structural changes to programs. The latter also provides access to the execution context of a program, which can then be manipulated to influence the normalization behavior of a program. Adding structural and procedural reflection to a programming language requires embedding a self representation of the language in the language, which is essentially a model of its implementation.

In relation to Smith's model of computation, the way to add structural reflection to any programming language is by extending the language with constructs for denoting and manipulating elements in the internal domain. The parts of the implementation that constitute the internal domain can be identified by looking for the structures in the implementation that implement the possible outcomes of internalization and normalization. So in other words, we should extend the language with means to denote the internal representation of numbers, characters, procedures, and so forth. Adding behavioral reflection requires providing means to influence the normalization process, e.g. by making it possible to call the interpreter or compiler from within the language.

3 Embedding a Model of Lisp in Lisp

One of the most difficult things about reflection to deal with is the fact that it "messes up" the way programmers think. It does so because reflection strips away the abstract model offered by the programming language. When using reflection, the programmer is thinking in terms of the language's implementation, and no longer in terms of the programming model behind the language. In a way, learning about reflection is similarly shocking as it was finding out that Santa Claus isn't real, but that it is your parents who give you your presents. When you first found out about this, you were very upset and you resented your parents for putting up a charade like that. After a while, though, you came to realize that it was actually nice of them to introduce you to the illusion of a Santa Claus, and that it is in fact still nice. Once you know your parents are the ones giving the presents, you can even influence them to make sure you get the gifts you really want. Similarly, reflection allows you to influence the language implementation to make sure you get the programming language you really want.

In what follows, we illustrate how to turn Lisp into a reflective variant. To be more precise, we turn the "prototypical" Lisp or "ProtoLisp" into its reflective variant. Here, ProtoLisp is what we like to call the subset of Lisp presented in textbooks to learn about implementation strategies for Lisp [5].

3.1 The World in Terms of ProtoLisp (Externalization in ProtoLisp)

The ProtoLisp programmer thinks in terms of s-expressions, evaluation and procedure application. An s-expression constitutes a program to be evaluated by the computer. The syntax of s-expressions is parenthesized prefix notation: S-expressions look like lists of symbols. The first symbol designates an operator or *procedure*, while the rest designates the operands or *arguments*. For example, the s-expression `(+ 1 2)` has an equivalent mathematical notation `1 + 2` and evaluates to `3`. An important procedure in ProtoLisp is `lambda`, which can be used to create new procedures.[1] It takes two arguments: a list, representing the procedure's arguments, and a body, an s-expression. For example, `(lambda (x) (+ 1 x))` creates a procedure that takes one argument `x` and adds `1` to it.

There are more kinds of objects we can talk about in ProtoLisp. The categories of the kinds of such objects are the programming language's *types*. ProtoLisp's types include numbers, truth values, characters, procedures, sequences, symbols and pairs (a pair denotes a procedure call).[2] The procedure `type` can be used to get hold of the type of a particular object.

In reality, the interpreter does not know about "numbers" or "procedures" or any of those other types. The programmer only thinks that what the interpreter shows him is a number or a procedure, though in reality, it shows him something completely different, like instances of classes implementing closures, arrays of characters representing strings, and other bits and bytes. We only choose to treat them as numbers and procedures and characters and so on. A language's types, the built-in operators, the way objects are printed, is what implements externalization. Reflection provides a look at the *actual* implementation of those objects. To know what exactly constitutes the implementation of such objects, we have to delve into the implementation of the ProtoLisp interpreter.

3.2 Internalization in ProtoLisp

ProtoLisp is an interactive language, implemented in this paper in Common Lisp using the Common Lisp Object System (CLOS). It consists of an endless read-evaluate-print-loop (repl), which repeatedly asks the programmer to type in an s-expression, performs evaluation of the s-expression and prints the result of the evaluation. The code for this is depicted in Fig. 2. In the code, we use the terminology by Smith: so *normalize* and *internalize* instead of the more traditional terms *evaluate* and *parse*. Note that the interpreter is implemented in continuation-passing style, meaning the control-flow is implemented by explicitly passing around "continuations". Continuations are implemented here as functions that take one argument. The third argument passed to `normalize` is a function that encodes the continuation of what needs to happen after an s-expression is evaluated, namely printing the result and starting the loop again.

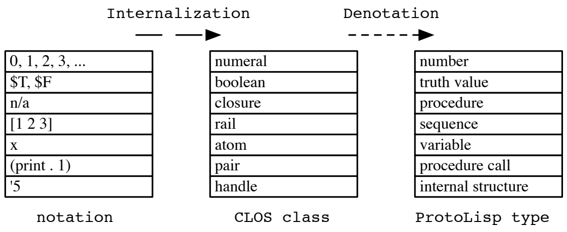
A program is initially just a string of characters. The ProtoLisp interpreter needs to parse a program string to something structured before it can be manipulated for evaluation. This structure is what comes out of the call to `prompt&read` in Fig. 2 (the first argument to `normalize`), which calls `internalize` to create that structure. From now on we refer to instances of those classes as *internal*

[1] In ProtoLisp, we call `lambda` a procedure because we do not distinguish between procedures and special forms, as is traditionally done in Lisp dialects.

[2] In this paper, we use the term *type* in the tradition of dynamically typed languages, where values are tagged with runtime type information. Various Lisp dialects differ in the diversity of the types they provide.

```
(defun read-normalize-print ()
  (normalize (prompt&read) *global* (lambda (result!)
                                      (prompt&reply result!)
                                      (read-normalize-print))))

(defun prompt&read ()
  (print ">")
  (internalize (read)))
```

Fig. 2. The ProtoLisp read-eval-print loop or repl



notation	CLOS class	ProtoLisp type
0, 1, 2, 3, ...	numeral	number
$T, $F	boolean	truth value
n/a	closure	procedure
[1 2 3]	rail	sequence
x	atom	variable
(print . 1)	pair	procedure call
'5	handle	internal structure

Fig. 3. Mapping notations to internal structures

structures. Depending on what the program looks like, a different internal structure is created. For example, if what is read is a digit, then an instance of the class `numeral` is created, if what is read is something that starts with a left-brace `(` and closes with a right brace `)`, then an instance of the class `pair` is created, and so on. In ProtoLisp, there is a syntax specific for each of the language's types. The `internalize` function dispatches on that syntax and creates instances of the appropriate classes.

Fig. 3 aligns all of the ProtoLisp types (third table) with examples showing their notation (first table). The second table lists the CLOS classes to which those notations internalize. Note that there is no specific syntax for denoting procedures. They are created by `lambda`, as discussed in the previous section.

3.3 Normalization in ProtoLisp

The `normalize` function depicted in Fig. 4 is the heart of the ProtoLisp interpreter. It implements how to simplify an s-expression to its normal form, which is something that is self-evaluating, like a number. The `normalize` function takes three arguments, namely an *s-expression*, an *environment* and a *continuation*:

– The s-expression is the ProtoLisp program to be evaluated. It is an instance of any of the classes listed in the second table of Fig. 3.
– The environment parameter is needed to keep track of the *scope* in which `normalize` is evaluating an s-expression. Environments are instances of the class `Environment` and map variables to bindings. There are two functions defined for environments: `binding` takes an environment and an atom, and returns the object bound to the atom in the given environment. `bind` takes an environment, an atom and an internal structure, and creates a binding in the

environment, mapping the atom to the internal structure. The `normalize` function is initially called with a *global environment*, which is bound to the variable `*global*` and provides bindings for all primitive procedures defined in the language (like `+`).

– Since the interpreter is written in continuation-passing style, the control-flow is explicitly managed by passing around *continuations*. Continuations are implemented as Common Lisp functions that take one argument, which must be an internal structure. As an example of a continuation, consider the `lambda` form in the source code of the `read-normalize-print` function in

```
;structure = numeral | boolean | closure | rail | atom | pair | handle

;normalize: structure, environment, function -> structure

01. (defun normalize (exp env cont)
02.   (cond ((normal-p exp) (funcall cont exp))
03.         ((atom-p exp)
04.           (funcall cont (binding exp env)))
05.         ((rail-p exp)
06.            (normalize-rail exp env cont))
07.         ((pair-p exp)
08.            (reduce (pcar exp) (pcdr exp) env cont))))

;normalize-rail: rail, environment, function -> rail

11. (defun normalize-rail (rail env cont)
12.   (if (empty-p rail)
13.     (funcall cont (rcons))
14.     (normalize (first rail)
15.                env
16.                (lambda (first!)
17.                  (normalize-rail (rest rail)
18.                                  env
19.                                  (lambda (rest!)
20.                                    (funcall cont (prep first! rest!))))))))

;reduce: atom, rail, environment, function -> structure

21. (defun reduce (proc args env cont)
22.   (normalize proc env
23.              (lambda (proc!)
24.                (if (lambda-p proc!)
25.                  (reduce-lambda args env cont)
26.                  (normalize args env
27.                             (lambda (args!)
28.                               (if (primitive-p proc!)
29.                                  (funcall cont (wrap (apply (unwrap proc!) (unwrap args!))))
30.                                  (normalize (body proc!)
31.                                             (bind-all (pattern proc!)
32.                                               args! (environment proc!))
33.                                             cont))))))))

;reduce-lambda: rail, environment, function -> closure

41. (defun reduce-lambda (args env cont)
42.   (let ((argument-pattern (first args))
43.         (body (second args)))
44.     (funcall cont
45.              (make-closure
46.                :body body
47.                :argument-pattern (make-rail :contents (unwrap argument-pattern))
48.                :lexical-environment env))))
```

Fig. 4. The continuation-passing-style interpreter for ProtoLisp

Fig. 2, which takes one argument `result!`, prints it on the screen and calls itself. This function is the continuation for the call to `normalize` in Fig. 2, which means that the function is called by the `normalize` function instead of returning to the caller.[3]

The `normalize` function distinguishes between four cases, the branches of the conditional form `cond` in Fig. 4. Depending on the type of the s-expression being normalized, a different evaluation strategy is taken.

An s-expression *in normal form* (`normal-p`) is self-evaluating. This implies that when normalizing it, it can simply be returned. This is shown in line 2 of Fig. 4, where the continuation of `normalize` is called with an s-expression in normal form. Examples of self-evaluating s-expressions are instances of the classes `numeral` and `closure`, which implement numbers and functions.

An atom, which denotes a variable (`atom-p`), is normalized by looking up its binding in the environment with which `normalize` is called, and returning this binding. Line 4 in Fig. 4 shows this: It displays a call to the continuation with the result of a call to `binding` as an argument. The latter searches the binding for the atom `exp` in the environment `env`.

A rail (`rail-p`) is normalized by normalizing all of its elements. Fig. 4 displays the source code for `normalize-rail`, which does this. It shows that a new rail is constructed out of the normalized elements of the original rail. `empty-p` is a function that checks whether a given rail has zero elements. `rcons` creates a new rail as an instance of the class `rail`. `first` and `rest` return the first element and all the other elements in a rail respectively. `prep` prepends an internal structure to a rail. So, for example, when one types `[1 (+ 1 1) 3]` in the repl, then `[1 2 3]` is displayed in return as the result of normalization.

A pair, denoting a procedure call (`pair-p`), is normalized by calling `reduce`[4], whose source code is also shown in Fig. 4. It is passed as arguments – besides the environment and the continuation of the call to `normalize` – the name of the procedure being called and the procedure call's argument list. They are obtained by calling the functions `pcar` and `pcdr` on the pair respectively (see line 8).

The source code of the `reduce` function, which is of course also written in continuation-passing style, is listed in Fig. 4 as well. Through a call to `normalize`, it looks up the binding for the procedure call's name `proc` (an atom). The continuation of that `normalize` call implements the rest of the `reduce` logic. It gets called with an instance of the class `closure`, which implements procedures (see below). If the latter closure represents the `lambda` procedure, checked on line 24 using the predicate `lambda-p`, then `reduce-lambda` is called, otherwise the rail of arguments is normalized, and depending on whether the closure represents a primitive procedure or not, `reduce` proceeds appropriately.

In case the procedure being called is a primitive procedure, as checked with `primitive-p` on line 28 in Fig. 4, a procedure call is interpreted by deferring it

[3] We assume that our implementation language Common Lisp supports tail recursion. Smith shows that this is ultimately not necessary, since all recursive calls are in tail position in the final 3-Lisp interpreter, resulting in a simple state machine [4].

[4] `reduce` is traditionally called `apply`.

to the Common Lisp implementation. Using `unwrap`, the closure object bound to `proc!` is translated to a Common Lisp function that implements the same functionality. This function is called with the arguments of the procedure call (bound to `args`) after mapping them to matching Common Lisp values. The result of this call is returned after turning it into an equivalent internal structure again by means of `wrap`, since the ProtoLisp interpreter is defined only for such structures. The details of `wrap` and `unwrap` are discussed in a following section.

Finally, when the procedure being called is user-defined, `reduce` proceeds by normalizing the procedure's body with respect to the procedure's environment, extended with bindings for the procedure's variables (see lines 30–32).

`reduce-lambda`, also listed in Fig. 4, takes as arguments the rail of arguments of the `lambda` procedure call being normalized, an environment and a continuation. It creates an instance of the class `closure` (see `make-closure` on line 45), where the slots `body`, `argument-pattern` and `lexical-environment` are bound to a body, a rail of variables and the environment passed to `reduce-lambda` respectively. The body is what is obtained by selecting the second element in `args`, while the argument pattern is obtained by selecting the first element from `args`. For example, when reducing the procedure call denoted by `(lambda (x) (+ x 1))`, then upon calling `reduce-lambda`, `args` will be bound to the rail denoted by `[(x) (+ x 1)]`, `body` will be bound to the rail denoted by `[(+ x 1)]` and `argument-pattern` to `[x]`.

3.4 Structural Reflection in ProtoLisp

Knowing the implementation of ProtoLisp, we are able to extend the language with reflection. With reflection we should be able to program at the level of the ProtoLisp implementation by writing ProtoLisp programs. To this end, the ProtoLisp language needs to be extended in such a way that it provides the illusion as if the implementation were written in ProtoLisp itself (and not in Common Lisp). In this section, we discuss how to extend ProtoLisp with structural reflection, while the next section discusses adding procedural reflection. ProtoLisp extended with structural reflection is dubbed *2-Lisp* by Smith.

Adding structural reflection requires two extensions: Firstly, we need to extend the ProtoLisp language with the ADTs that make up the ProtoLisp implementation. Secondly, we also need a way to get hold of the internal structure of a ProtoLisp value. The ProtoLisp implementation consists of a number of CLOS classes implementing the ProtoLisp types, and Common Lisp functions to manipulate instances of these classes. The classes are listed in the second table of Fig. 3 and the various functions, such as `normalize`, `reduce`, `pcar`, `binding` and so on, are discussed in the previous section. For example, the CLOS class `pair` and the functions `pair-p`, `pcar` and `pcdr` implement an ADT for representing procedure calls that needs to be added to the ProtoLisp language as a corresponding ProtoLisp type and corresponding ProtoLisp procedures.

Similarly to adding a `pair` type and procedures that work for pairs, we extend ProtoLisp with types and procedures that mirror the CLOS classes implementing the rest of the ProtoLisp types, like numbers, characters, procedures and so on.

Some of the implementation functions we need to port to ProtoLisp require types of arguments the ProtoLisp programmer normally does not deal with. The `normalize` function, for example, takes an environment parameter. We need to add types and procedures to ProtoLisp for these classes as well.[5]

We also add `'` to ProtoLisp, which is syntax for denoting the internal structure of an s-expression that is created when the s-expression is internalized.[6] For example, `'(+ 1 1)` denotes the instance of the class `pair` that is obtained when internalizing `(+ 1 1)`. The result can be used as a pair: For example, `(pcar '(+ 1 1))` returns `'+` and `(pcdr '(+ 1 1))` returns `'[1 1]`.

`'` allows getting hold only of internal structures of values with corresponding syntax. So for example, `'` does not allow accessing the internal structure of a procedure, because a procedure cannot be created via internalization alone (see Fig. 3). We add the procedure `up` to ProtoLisp that normalizes its argument and then returns its internal structure. Thus typing in `(up (lambda (x) (+ 1 x)))` will return a closure. The `down` procedure is added to ProtoLisp to turn an internal structure obtained via `'` or `up` into a ProtoLisp value. E.g. `(down '3)` returns the "number" 3.

We can now, for example, define a procedure that prints a pair in infix notation. The definition is shown in the following session with the repl:

```
> (set print-infix (lambda (structure)
                     (if (pair-p structure)
                       (begin
                         (print "(")
                         (print-infix (1st (pcdr structure)))
                         (print (pcar structure))
                         (for-each print-infix (rest (pcdr structure)))
                         (print ")"))
                       (print (down structure)))))
<procedure:print-infix>
> (print-infix '(+ 1 1))
(1 + 1)
```

The `print-infix` procedure checks whether its argument is a pair. If it is a pair, the first argument to the procedure call is printed, followed by the procedure's name and the rest of the arguments. Otherwise, it is just printed.

Apart from inspecting internal structures, we can also modify them. For example, the following code shows how to add simple before advice to closures.

```
> (set advise-before
    (lambda (closure advice)
      (set-body closure (pcons 'begin (rcons advice (body closure))))))
> (set foo (lambda (x) (+ x x)))
> (foo 5)
10
> (advise-before (up foo) '(print "foo called"))
> (foo 5)
foo called
10
```

[5] A complete listing of these procedures is beyond the scope of this paper. The interested reader is referred to the original 3-Lisp reference manual [4].

[6] `'` is called "handle", in contrast to Common Lisp's and Scheme's "quote", also abbreviated as `'`.

The `advise-before` procedure changes the body of a closure to a sequence that first executes a piece of advice and then the original closure body.

3.5 Procedural Reflection in ProtoLisp

As a final extension we add procedural reflection to ProtoLisp. This extension is dubbed *3-Lisp* by Smith. The goal of procedural reflection is to allow the programmer to influence the normalization process. To this end, ProtoLisp is extended with *reflective lambdas* to access a program's execution context, that is the interpreter's temporary state, which consists of the expression, the environment and the continuation at a particular normalization step. A reflective lambda looks like a regular procedure, with the exception that the length of its argument list is fixed: `(lambda-reflect (exp env cont) (cont (down (1st exp))))`. A reflective lambda has three parameters. When a reflective lambda call is resolved, they are bound to the expression, the environment and the continuation at that normalization step.

As a first example, consider implementing a `when` construct. This construct is similar to `if`, but `when` has only one consequential branch. The code below shows how to implement it in 3-Lisp.

```
(set when (lambda-reflect (exp env cont)
            (normalize (cons 'if (up [ (down (1st exp))
                                       (down (2nd exp))
                                       $F ]))
                       env cont)))
```

`when` is defined as a reflective lambda. When the interpreter normalizes a pair with `when` as procedure name, it transforms the pair into an `if` pair and normalizes that one instead. The body of the `when` construct consists of a call to `normalize`. The first argument is the `if` pair that is constructed out of the `when` pair, the second and the third argument are just the same environment and continuation for normalizing the `when` pair. For example, the following two expressions are equivalent:

```
(when (= (mod nr 1000) 0)
  (print "Congratulations! You win a prize."))

(if (= (mod nr 1000) 0)
  (print "Congratulations! You win a prize.")
  $F)
```

As a second example, consider one wishes to implement a `search` procedure. It takes as arguments a test and a list of elements, and returns the first occurrence in the list that satisfies the test. For example:

```
>(search number-p [ 'licensed 2 'kill ])
2
```

A procedure like `search` can easily be implemented, but consider that in doing so, we want to reuse the library procedure `for-each`, which applies a given procedure to *each* element in a given list:

```
(set for-each (lambda (f lis)
                (when (not (empty lis))
                  (begin (f (1st lis)) (for-each f (rest lis))))))
```

The code below shows an implementation of `search` in terms of `for-each`. The trick is to implement `search` as a reflective lambda: We call `for-each` with the test passed to `search`, and when this test succeeds, it calls the continuation `cont`. This is the continuation of normalizing the `search` pair: As such, we jump out of the `for-each` loop as soon as an element satisfying the test is found.

```
(set search (lambda-reflect (exp env cont)
              (normalize (2nd exp) env
                (lambda (list!)
                  (for-each (lambda (el)
                              (when ((down (binding (1st exp) env)) el)
                                (cont el)))
                            (down list!))))))
```

3.6 Implementing Reflection

Meta types and procedures. Adding reflection to ProtoLisp requires extending the language with types and procedures that mirror the ADTs making up its implementation. In our implementation of ProtoLisp, the latter ADTs are implemented using CLOS classes and functions. For the better part, porting them to ProtoLisp is a matter of extending the language with new *primitive* procedures and types that simply wrap the corresponding CLOS classes and functions.

For example, in Fig. 5 we list definition skeletons of the class `pair` and its functions `pcar` and `pcdr`. The code also shows how we extend ProtoLisp's global environment with a definition for the ProtoLisp procedure `pcar`. The latter definition is implemented as a closure tagged "primitive", with a reference to the Common Lisp function `pcar` as its body. So when the procedure `pcar` is called in ProtoLisp, the interpreter recognizes it as a "primitive" closure object, defers the call to the Common Lisp implementation, and ultimately calls the Common Lisp function stored in the closure object.

Wrapping and unwrapping. Recall line 29 of the `reduce` function in Fig. 4, which handles calls to primitive procedures: A Common Lisp function is fetched from the primitive closure object and then called with the provided arguments. However, it is necessary that these arguments are first mapped onto "equivalent" Common Lisp values by means of `unwrap`. For example, given an instance of the class `numeral`, `unwrap` returns the equivalent Common Lisp `number`.

When there is a one-to-one mapping between simple Common Lisp and ProtoLisp values, like between ProtoLisp numbers and Common Lisp numbers, the

```
(defclass pair (handle) ...)
(defun pcar (p) ...)
(defun pcdr (p) ...)

(defvar *global* (make-environment))

(bind *global* (make-atom :name (quote pcar))
               (make-closure :ctype *primitive-tag*
                             :body (function pcar)
                             :lexical-environment *global*))
```

Fig. 5. Extending ProtoLisp with procedures mirroring the implementation functions

```
(defmethod unwrap ((closure closure))
  (cond ((primitive-p closure) (body closure))
        ((reflective-p closure)
         (make-function :closure closure
                        :lambda (lambda (&rest args)
                                  (error "Cannot call reflective closure from CL."))))
        (t
         (make-function :closure closure
                        :lambda (lambda (args)
                                  (reduce closure
                                          (make-rail :contents (list args))
                                          *global*
                                          (lambda (result!)
                                            (prompt&reply result!)
                                            (read-normalize-print)))))))))
```

Fig. 6. Unwrapping closure objects to Common Lisp functions

implementation of `unwrap` is straightforward. Unwrapping a closure which is tagged "primitive" is also straightforward: In this case, `unwrap` simply returns the Common Lisp function stored as the closure's body. Unwrapping other closures is more complicated, because their bodies contain ProtoLisp source code that Common Lisp functions cannot directly deal with, so they need to be handled specially when Common Lisp code wants to call them directly. Therefore, such closures are unwrapped by creating a special function object, which stores a reference to the original closure and a Common Lisp function handling it (see Fig. 6). Common Lisp code that wants to call such function objects needs to invoke the stored function instead.[7] What special action is performed by such a function depends on whether the original closure is tagged as "non-reflective" or "reflective".

When `unwrap` is called with a "non-reflective" closure, we can map it to a Common Lisp function that simply invokes the ProtoLisp interpreter by calling `reduce` with the closure and the arguments received by that Common Lisp function. We also need to pass an appropriate environment and continuation to `reduce`. Since the arguments passed to the unwrapped procedure are already normalized, it does not really matter which environment to pass, so passing the global environment here is as good as any other choice. We can answer the question which continuation to pass by making the following observation: In our implementation of ProtoLisp, the only higher-order primitive procedures that take other procedures as arguments and eventually call them are `normalize` and `reduce`, which are the two main procedures of the interpreter and receive procedures as their continuation parameters. See, for example, the call to `normalize` in the definition of `search` in the previous section, which receives a non-reflective ProtoLisp procedure as a continuation. These continuation procedures are expected to call other continuations, which will ultimately end up in displaying a result in the repl and waiting for more s-expressions to evaluate, because the repl is where *any* evaluation originates from. However, to ensure that the repl is not accidentally exited by continuation procedures which simply return a value – for

[7] In our implementation, we have used *funcallable objects*, as provided by CLOS [3], to avoid having to update all the places in the code where functions are called.

example when calling `(normalize '(+ 1 1) global (lambda (res) res))` – we pass a "fallback" continuation to `reduce` that simply ends up in the repl as well (see Fig. 6).[8]

We stress, however, that unwrapping "non-reflective" closures in this way is based on the assumption that the only higher-order primitive procedures which call their procedure arguments are indeed `normalize` and `reduce`. If we want to provide other higher-order procedures as primitives, like for example `mapcar`, we need to pass other environments and continuations in `unwrap` as special cases. Fortunately, this is not necessary because such higher-order procedures can always also be implemented in ProtoLisp itself.[9]

As a final case in `unwrap`, we need to consider how to map closures tagged as "reflective" onto something appropriate in Common Lisp. However, there is no way we can map a reflective closure to a Common Lisp function with the same behavior because it would require that there are already similar procedurally reflective capabilities available in Common Lisp itself, which is not the case. Therefore, we just ensure that calling an unwrapped reflective closure signals an error. We can still pass reflective procedures as arguments to primitive ProtoLisp procedures, but only to eventually receive them back in ProtoLisp code again, where they have well-defined semantics. Since `normalize` and `reduce` are the only higher-order primitive procedures in ProtoLisp that actually call their procedure parameters, this is no big harm: We consider the possibility to reflect on the program text, the environment and the continuation at some arbitrary implementation-dependent place in the interpreter to be highly questionable.[10]

The inverse of the `unwrap` function is the `wrap` function, which maps a Common Lisp value to the corresponding ProtoLisp value (an internal structure). The `wrap` function is used to turn the result of interpreting a primitive procedure into a proper internal structure (see line 29 in Fig. 4). For example, given a Common Lisp number, `wrap` returns an instance of the class `numeral`. When passed a Common Lisp list, `wrap` returns a rail object. Given a Common Lisp function, `wrap` returns a closure object which is tagged "primitive". Given a function object wrapping a closure, `unwrap` returns the closure. Similarly `wrap` maps other Common Lisp values to appropriate internal structures.

Up, down & '. As discussed in the section on structural reflection, ProtoLisp is extended with `'`, `up` and `down` for denoting internal structures. The procedure `up` returns the internal structure of its argument, and is implemented as a primitive procedure (see above) that calls the function `wrap`. The procedure `down`, which returns the ProtoLisp value matching the given internal structure, is

[8] If ProtoLisp is not used as a repl, but for example as an extension language inside other applications, we have to use other "fallback" continuations here, which would simply return the value to the original call site.

[9] Primitive higher-order procedures may be interesting for efficiency reasons, though.

[10] Note, however, that passing reflective procedures to the underlying implementation can be supported by a "tower" of interpreters, and is actually one motivation for the notion of reflective towers, which we discuss in the next section.

```
(defun reduce-reflective (proc! args env cont)
  (let ((non-reflective-closure (de-reflect proc!)))
    (normalize (body non-reflective-closure)
               (bind-all (lexical-environment non-reflective-closure)
                         (argument-pattern non-reflective-closure)
                         (make-rail :contents (list args env cont)))
               (lambda (result!)
                 (prompt&reply result!)
                 (read-normalize-print)))))
```

Fig. 7. Interpreting calls to `lambda-reflect`

implemented as a primitive procedure that calls the function `unwrap`. `'` is syntax added for returning the result of internalizing a ProtoLisp s-expression.

`wrap` and `unwrap` also work with instances of the class `handle`. Handle objects "wrap" structures that are already internal, by just storing references to the wrapped objects. If `wrap` receives an internal structure, it just wraps it in a handle object. If `unwrap` receives an instance of the class `handle`, it returns whatever the handle object holds.

Lambda-reflect. We also extend ProtoLisp with `lambda-reflect` to render the temporary state of the interpreter explicit. As discussed, `lambda-reflect` resembles `lambda`: When a call to `lambda-reflect` is interpreted, a closure is created that is tagged "reflective". When a call to a reflective procedure is interpreted, it is turned into an equivalent non-reflective procedure which is called instead. Furthermore, that procedure is passed the s-expression, the environment and the continuation with which the interpreter was parameterized when interpreting the reflective procedure call.

In the implementation, we extend `reduce` with a case for recognizing reflective procedure calls and passing them on to `reduce-reflective`, whose code is shown in Fig. 7. The function `de-reflect` turns a reflective closure into a regular closure by just changing its tag from "reflective" to "non-reflective". Apart from that, it has the same argument pattern, the same lexical environment and the same body as the original reflective closure. The procedure `bind-all` extends the lexical environment of the closure by mapping its argument pattern to the (unevaluated) arguments, the environment, and the continuation with which `reduce-reflective` is called. Finally, the call to `normalize` triggers evaluating the body of the closure in terms of the extended environment. The continuation passed to `normalize` is the repl's continuation from Fig. 2. When the body of the reflective procedure contains a call to the continuation it receives as an argument, then the continuation in Fig. 7 will actually never be called. However when the body of the reflective procedure does not call its continuation, then the repl continuation will be (implicitly) called. Again, this is to avoid that the repl is accidentally exited (like when unwrapping non-reflective closures).

3.7 The Tower Model

One of the most debated ideas of Smith's account on reflection is the tower model. In this model, 3-Lisp is implemented as an infinite stack of meta circular interpreters running one another. This solves some conceptual and technical problems that arise when implementing procedural reflection in 3-Lisp. One such problem is the following: Consider a reflective lambda where the continuation passed to it is ultimately not called. In a naive implementation of 3-Lisp, evaluating a call to such a reflective lambda would result in exiting the 3-Lisp repl and falling back to the implementation level, since the passed continuation includes the continuation of the ProtoLisp repl. Generally speaking, this is not the desired functionality. Using an interpreter that implements the tower model, calling a reflective lambda can be resolved by the interpreter that is running the repl the programmer was interacting with at the time of the call. The only way the programmer will get back to the original interpreter is when the continuation is called inside the reflective lambda's body. When the continuation is not called, then the programmer just stays in the upper interpreter. This is referred to as "being stuck" at a meta level [4]. As such there is no danger the programmer falls back to the implementation level [6].

Problems like those can occur when there is *reflective overlap*, i.e. when a reflective language construct renders some part of the implementation explicit, but also relies on it [2]. In our example, reflective lambdas render the continuation explicit, but when it is not called inside its body, then the entire interpretation process is stopped. Tower models are generally believed to solve problems introduced by reflective overlap. However, they are not the only solution. The Scheme language, for example, is not designed as a tower, and though it introduces `call/cc` to capture a continuation, not calling it will not result in exiting Scheme, but it will just be implicitly called, no matter what. Our implementation behaves similar in that respect.

3.8 Summary – A Recipe for Reflection

In the previous sections, we extended ProtoLisp with reflection, which makes it possible to program at the level of the ProtoLisp implementation by writing ProtoLisp programs. The steps we took for adding reflection to ProtoLisp can be synthesized to a recipe for adding reflection to *any* programming language.

There are two parts to implementing structural reflection. First of all, one needs to identify the ADTs in the implementation that are used for representing programs, and expose them in the language itself. Secondly, one needs to equip the language with a mechanism to turn programs into first-class entities. Identifying which ADTs are potential canditates for structural reflection is done by looking at the possible outcomes of internalization (parsing) and normalization (evaluation). Note that in porting the ADTs, we need to make it appear as if they were implemented in the language itself. For adding structural reflection to ProtoLisp, this means adding new primitive types and procedures wrapping the ADTs that implement characters, numbers, procedures, and so on. The procedure `up` and the `'` syntax enable getting hold of the internal representations of programs. The procedure `down` turns programs into regular values again. Such mechanisms are implemented by means of a *wrap/unwrap* mechanism that tags internal structures in a way that ensures that the programmer can interface them using the wrapped implementation procedures.

Procedural reflection is implemented by adding mechanisms that allow the programmer to influence the normalization of a program at well-defined steps. This includes pausing the normalization, inspecting and changing the processor state at that time, and continuing the normalization. In ProtoLisp, we added `lambda-reflect`, which allows defining reflective procedures. A reflective procedure is passed the state of the processor (an expression, an environment and a continuation), which can be manipulated as regular data inside its body. To proceed with the computation, the programmer needs to set the interpreter state by calling the `normalize` procedure. In our implementation, `lambda-reflect` is implemented as a special case in the interpreter, and `normalize` as a primitive procedure that calls its counterpart in the actual implementation.

In the next section, we give an overview of some reflective programming languages, including both historical and contemporary approaches. Such approaches are all more or less compatible with the recipe outlined above.

4 Related Work

Historical overview. Reflective facilities were already part of Lisp before Smith introduced the concept of procedural reflection. Lisp 1.5 [7] provides a quote mechanism that allows constructing Lisp programs on the fly at runtime. For example, the Lisp 1.5 expression `(let ((x 1)) (apply '(lambda (y) (+ x y)) (list 2)))` returns 3: A list whose first element is the symbol `lambda` is interpreted as a function, and since Lisp 1.5 is a dynamically scoped Lisp dialect, the variable references see the dynamic bindings of the respective variables even in the code constructed on the fly. (The quoted lambda expression in the example may as well be the result of a computation.) This ability to construct and execute code on the fly corresponds to Smith's notion of structural reflection, where procedures can be constructed and manipulated via `up` and `down`. Lisp 1.5 also provides the ingredients of procedural reflection: An `fexpr` is a function that gets unevaluated arguments passed as program text, and it is possible to define `fexpr`s as user programs. The `alist` provides access to the environment mapping variables to values,[11] and the "push down list" is the call stack (continuation) that can also be accessed from within Lisp programs. Taken together, these features correspond to the notion of reflective procedures in 3-Lisp. For example, such reflective features were used to introduce the concept of advice [8].

Unfortunately, dynamic scoping is problematic when it is the default semantics for Lisp. Especially it leads to the so-called "upward" and "downard funarg problems" [9]. While they can be solved by dynamic closures and spaghetti stacks to a certain degree, only the introduction of lexical closures in Scheme fully solved all aspects of the underlying issues [10]. Lexical closures got picked up in Common Lisp and most other Lisp dialects thereafter. However, lexical closures make some of the aforementioned reflective features of Lisp 1.5 less straightforward to

[11] In Lisp 1.5, only one such environment exists.

integrate as well. 3-Lisp can be regarded as a reconceptualization of such reflective facilities in the framework of a lexically scoped Lisp dialect.

Current Lisp dialects, among which Scheme and Common Lisp are the most widely used ones, typically provide only restricted subsets of structural reflection: Scheme's `eval` and Common Lisp's `eval` and `compile` can be used to turn a quoted lambda expression into a function (similar to `down`), but they cannot be enclosed in arbitrary lexical environments, only in global or statically predefined environments. There is also typically no construct corresponding to `up` available that would allow retrieving the original definition of a function. In terms of procedural reflection, neither Scheme nor Common Lisp allow defining functions that receive unevaluated arguments as program text, neither Scheme nor Common Lisp specify operators for reifying lexical environments, and only Scheme provides `call/cc` for reifying the current continuation. Macros were introduced into Lisp 1.5 in the 1960's [11], and are considered to be an acceptable and generally preferrable subset of reflecting on source code [12]. The difference in that regard to reflective procedures, `fexpr`, and so on, is that macros cannot be passed around as first-class values and are typically restricted from accessing runtime values during macro expansion. This allows compiling them away before execution in compiled systems, as is mandated for example by current Scheme and ANSI Common Lisp specifications [13,14]. Useful applications of first-class lexical environments in Scheme have been described in the literature [15,16], but the only Scheme implementation that seems to fully support first-class environments at the time of writing this paper is Guile, and the only Lisp implementation that seems to do so is clisp in interpreted mode.[12]

Wand and Friedman were the first to follow up on Smith's ideas, and in their research they concluded that the tower model unnecessarily introduces extra complexity to reflective programming. One of their first results is Brown, a reflective variant of Scheme, which shows that implementing procedural reflection does not require a tower architecture [17]. In their explanations, Wand and Friedman introduced the now widely used term *reification*, which is the process of turning implementation structures into first class representations. However, they differ from 3-Lisp by renouncing an explicit `up/down` mechanism. Brown instead relies on Scheme's quoting facilities as a "reification" mechanism, and the `up/down` mechanism is dismissed as a "philosophical concern". However, as we discussed in Section 3.4, this mechanism is necessary for being able to denote the internal representation of program values that cannot be identified by a string of source text alone. Hence, because the `up/down` mechanism is missing, Brown's reification mechanism is restricted: It is for example impossible in Brown to get hold of a closure. The contribution in this paper is the integration of an explicit `up/down` mechanism with a tower-less implementation of 3-Lisp, and in doing so, we introduced a correct semantics for wrapping and unwrapping reflective functions (see Section 3.6).

[12] See http://www.gnu.org/software/guile/guile.html and http://clisp.cons.org/. Some other Scheme implementations as well as the OpenLisp implementation of ISLISP claim to support first-class environments as well, but failed in some of our tests.

While solving some problems, the tower also introduces new problems. Suppose, for example, that the programmer modifies the `normalize` procedure so that its calls are logged. If the `normalize` procedure is identical for all of the interpreters in the tower, this implies that each call to `normalize` produces an infinite number of logs. To deal with this, the Blond language, based on 3-Lisp, is implemented as a tower where each interpreter (potentially) has a separate environment [18]. Finally, Asai et al. subsequently proposed a compilation framework based on partial evaluation for implementing the tower model efficiently [19].

Analysis of reflective programming models. There exist a great deal of programming languages offering reflective facilities. Depending on the kinds of reflective facilities they offer, and how these are implemented and used, we can classify these facilities as being a variant of either 2-Lisp or 3-Lisp. As we discussed previously, 2-Lisp offers the programmer structural reflection, while 3-Lisp grants procedural reflection. With structural reflection, it is possible to manipulate the internal representation of programs as if these were regular data. Using structural reflection, the programmer can make structural changes to programs, e.g. inline the code for specific function calls in a program or extend the body of a closure with code for caching. What is important to note is that structural changes to programs like these can be done without executing the program. As such, compilation techniques could be employed to implement structural reflection, which is in general believed to be more efficient than a dynamic implementation, as is required for implementing certain kinds of procedural reflection.

3-Lisp extends 2-Lisp by offering the programmer access to the execution context of a program. In 3-Lisp this execution context includes an environment and a continuation. This allows, for example, implementing one's own language constructs for exception handling or a module system as a 3-Lisp program. The execution context of a program is available only during program execution, and this requires an implementation that operates at runtime. In what follows, we give an overview of existing reflective programming languages. We briefly discuss their model and whether they offer structural or procedural reflection. We also investigate if the underlying model makes any explicit assumptions that exclude procedural reflection, e.g. for efficiency reasons.

The CLOS Metaobject Protocol (MOP) is a specification of how major building blocks of the Common Lisp Object System are implemented in terms of itself. It thus provides hooks to modify CLOS semantics by defining methods on subclasses of standard metaclasses. Such meta-level methods get invoked by conforming CLOS implementations, for example during the construction of class and generic function objects, and during slot access and method dispatch. The CLOS MOP is thus a reflective architecture, and is based on the model of procedural reflection in the sense that the hooks get triggered at runtime, while these aspects of a CLOS implementation are actually performed. As such, user-defined meta-level methods can make their results depend on runtime values. The CLOS MOP itself does not provide first-class representations of the execution context (like the current environment or the current continuation), but these could in principle be provided as orthogonal features. Later metaobject protocols tried to push the boundaries closer to structural reflection by computing more and more aspects of the object system before they get used, leading over hybrid systems like Tiny CLOS, towards load-time [20] and compile-time metaobject protocols [21], where all aspects are computed before they are ever used. At closer inspection, one can notice that the CLOS MOP is already a hybrid system, although with an emphasis on procedural elements.

The CLOS MOP can be understood as a combination of procedural reflection as in 3-Lisp together with Smalltalk's approach to object-oriented programming, where everything is an instance of a class, including classes themselves. Smalltalk's metaclasses provide a form of structural reflection, which for example allows manipulating method dictionaries, but lack meta-level protocols that can be intercepted in a procedurally reflective way (with the handling of the "message not understood" exception being a notable exception) [22]. However, Smalltalk provides first-class access to the current call stack via `thisContext`, which roughly corresponds to a combination of the environment and the continuation parameters in reflective lambdas [23]. In [24] Ducasse provides an overview of techniques, based on Smalltalk's reflective capabilities, that can be used to define a message passing control.

Self provides structural reflection via *mirrors* [25]. It can actually be argued that mirrors are a rediscovery of `up` and `down` from 2-Lisp, but put in an object-oriented setting. However, *mirrors* provide new and interesting motivations for a strict separation into internal and external representations. Especially, *mirrors* allow for multiple different internal representations of the same external object. For example, this can be interesting in distributed systems, where one internal representation may yield details of the remote reference to a remote object, while another one may yield details about the remote object itself. AmbientTalk/2 is based on mirrors as well, but extends them with *mirages* that provide a form of (object-oriented) procedural reflection [26].

Aspect-oriented programming [27] extends existing programming models with the means to "modify program join points". Depending on the aspect model at hand, program join points are defined as points in the execution of a program, or as structural program entities. In an object-oriented setting, examples of the former are "message sends" and "slot accesses", examples of the latter are classes and methods. The idea is that the programmer can make changes to program join points *without* having to change their sources, but by defining distinct program modules called "aspects". This property is called *obliviousness* and is believed to improve the quality of software in terms of better modularity.

One of the most influential aspect languages is AspectJ [28], which facilitates adding methods to classes, but also supports advising methods with logging code. Aspects are defined in terms of pointcut-advice pairs: Pointcuts are declarative queries over program join points, whereas advice consists of pieces of Java code that need to be integrated with the join points matched by a pointcut. AspectJ's pointcut language is a collection of predicates for detecting structural patterns in source code, like the names of classes or methods, where code needs to be inserted. AOP is a reflective approach in the sense that aspects are expressed as programs about programs, but unlike reflection, conventional AOP leaves out a model of the language implementation, which greatly reduces its expressiveness.

5 Conclusions

In this paper, we have given an overview of the notion of computational reflection, as introduced by Brian Smith in the early 1980's. His contributions include the notions of structural and procedural reflection, as well as a recipe for adding reflection not only to Lisp dialects, but to programming languages in general. We have a reconstructed an implementation of all the major elements of 3-Lisp, a procedurally reflective laguage, based on this recipe. We have chosen Common Lisp as the basis for our implementation, a modern Lisp dialect, which should make these contributions more accessible to a new and wider audience. We have finally used the terminological framework of computational reflection to analyze other metaprogramming and reflective approaches.

History shows that programming language designers always struggle to find the right balance between hiding language implementation details and making them accessible from within those languages. Apparently, it is inevitable that reflective features find their ways into programming languages, whether their designers are aware of this choice or not, because reflection is indeed a good compromise between hiding and revealing such details. On the other hand, designers and implementors have also always strived to minimize the cost of reflection by providing only subsets of such features. As we discussed in our overview of reflective programming models, most approaches offer only restricted subsets of structural reflection, because efficiency seems more straightforward to be achieved in this case using compilation techniques. Nevertheless, procedurally reflective features also always sneak in, but typically severely scaled down and poorly integrated with structural reflection. Due to such compromises, however, the complexity of writing reflective programs seems to increase, which in turn fortifies the view that reflection is 'dangerous' and should be provided in only small doses. Unfortunately, Smith's original account of computational reflection has been lost along the road.

We are convinced that it is time to rediscover and rethink reflection from the ground up, without any such compromises. Modern hardware is finally fast enough to effectively reduce the necessity to focus on efficiency alone. The rise in popularity of scripting languages in the last two decades shows that programmers are more and more interested in the flexibility offered by dynamicity and reflection, rather than the limiting constraints of staticity and encapsulation. To the contrary, we need more, not less reflection: For example, parallel programming models for multicore processors and distributed systems require new ways to toggle between absorbing and revealing details of language constructs, such that programs can better reason about events in the past, present and future of a computation in process. Brian Smith's notion of computational reflection is an invaluable basis to start from for these future investigations.

Acknowledgments

We thank Wolfgang De Meuter, Kris Gybels, Jorge Vallejos and the anonymous reviewers of S3 for their valuable comments on previous drafts of this paper. Charlotte Herzeel is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

1. Smith, B.C.: Reflection and semantics in lisp. In: POPL 1984: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 23–35. ACM Press, New York (1984)
2. Steyaert, P.: Open Design of Object-Oriented Languages. PhD thesis. Vrije Universiteit Brussel (1994)
3. Kiczales, G., Rivieres, J.D., Bobrow, D.: The art of the Metaobject Protocol. MIT Press, Cambridge (1991)
4. Smith, B.C., Rivieres, J.D.: Interim 3-Lisp Reference Manual. Intelligent Systems Laboratory. PALO ALTO RESEARCH CENTER. (June 1984)
5. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs, 2nd edn. MIT Press, Cambridge (1996)
6. Wand, M., Friedman, D.P.: The mystery of the tower revealed: a non-reflective description of the reflective tower. In: Proceedings of the 1986 ACM Symposium on LISP and Functional Programming, pp. 298–307 (August 1986)
7. McCarthy, J.: LISP 1.5 Programmer's Manual. MIT Press, Cambridge (1962)
8. Teitelman, W.: PILOT: A Step Toward Man-Computer Symbioses. PhD thesis, Massaschusetts Institute of Technology (1966)
9. Moses, J.: The function of function in lisp or why the funarg problem should be called the environment problem. SIGSAM Bull. 15, 13–27 (1970)
10. Steele, G.L., Sussman, G.J.: The art of the interpreter or, the modularity complex (parts zero, one, and two). Technical report, Cambridge, MA, USA (1978)
11. Hart, T.P.: Macro definitions for lisp. Technical report, Cambridge, MA, USA (1963)
12. Pitman, K.M.: Special forms in lisp. In: LFP 1980: Proceedings of the 1980 ACM conference on LISP and functional programming. ACM Press, New York (1980)
13. Sperber, M., Dybvig, R.K., Flatt, M., van Straaten, A., Kelsey, R., Clinger, W., Reese, J., Findler, R.B., Matthews, J.: Revised^6 report on the algorithmic language Scheme (September 2007)
14. Pitman, K.M., ed.: ANSI INCITS 226-1994 (formerly ANSI X3.226:1994) American National Standard for Programming Language Common LISP is the official standard (1994)
15. Queinnec, C., de Roure, D.: Sharing code through first-class environments. SIGPLAN Not. 31(6), 251–261 (1996)
16. Gelernter, D., Jagannathan, S., London, T.: Environments as first class objects. In: POPL 1987: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 98–110. ACM Press, New York (1987)
17. Friedman, D.P., Wand, M.: Reification: Reflection without metaphysics. In: Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pp. 348–355, (1984)

18. Danvy, O., Malmkjær, K.: A blond primer. Technical Report DIKU Rapport 88/21, DIKU (October 1988)
19. Asai, K., Masuhara, H., Matsuoka, S., Yonezawa, A.: Partial evaluation as a compiler for reflective languages. Technical report, University of Tokyo (1995)
20. Bretthauer, H., Davis, H.E., Kopp, J., Playford, K.J.: Balancing the EuLisp Metaobject Protocol. In: Proc. of International Workshop on New Models for Software Architecture, Tokyo, Japan (1992)
21. Chiba, S.: A metaobject protocol for C++. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1995), Austin, Texas, USA, October 1995. SIGPLAN Notices, vol. 30(10), pp. 285–299 (1995)
22. Brant, J., Foote, B., Johnson, R.E., Roberts, D.: Wrappers to the rescue. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 396–417. Springer, Heidelberg (1998)
23. Goldberg, A., Robson, D.: Smalltalk-80: The Language. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
24. Ducasse, S.: Evaluating message passing control techniques in smalltalk. Journal of Object-Oriented Programming (JOOP), 12(6), 39–44 (1999)
25. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: OOPSLA 2004: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 331–344. ACM Press, New York (2004)
26. Mostinckx, S., Cutsem, T.V., Timbermont, S., Tanter, É.: Mirages: behavioral intercession in a mirror-based architecture. In: DLS 2007: Proceedings of the 2007 symposium on Dynamic languages, pp. 89–100. ACM Press, New York (2007)
27. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
28. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)

A Source Code

```
;;; config.lsp

(defpackage 3-proto-lisp
  (:use common-lisp clos)
  (:shadow boolean atom length prep first rest nth body reduce)
  (:export read-normalize-print normalize-from-string)
  (:nicknames 3pl))

#|
  Copyright (c) 2007, 2008 Charlotte Herzeel

  Permission is hereby granted, free of charge, to any person
  obtaining a copy of this software and associated documentation
  files (the "Software"), to deal in the Software without
  restriction, including without limitation the rights to use,
  copy, modify, merge, publish, distribute, sublicense, and/or
  sell copies of the Software, and to permit persons to whom the
  Software is furnished to do so, subject to the following
  conditions:
```

```
  The above copyright notice and this permission notice shall be
  included in all copies or substantial portions of the Software.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
  EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
  OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
  NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
  HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
  WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
  FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
  OTHER DEALINGS IN THE SOFTWARE.
|#

;;; repl.lsp

(in-package 3-proto-lisp)

(defun read-normalize-print ()
  (declare (special *global*))
  (normalize (prompt&read)
             *global*
             (lambda (result!)
               (prompt&reply result!)
               (read-normalize-print))))

(defun prompt&read ()
  (format t ">")
  (three-lisp-read-and-parse))

(defun prompt&reply (result)
  (format t "~%~a~&" (print-to-string result)))

;; Normalize from string

;; Flag to see if in repl mode

(defparameter *repl-mode* T)

(defun normalize-from-string (string)
  (let ((*repl-mode* nil))
    (normalize (internalize (read-from-string string)) *global* (lambda (result!) result!))))

(defun create-meta-continuation ()
  (if *repl-mode*
      (lambda (result!)
        (prompt&reply result!)
        (read-normalize-print))
    (function unwrap)))

;; Loading ProtoLisp code from a file

(defun load-proto-lisp-file (filename-as-string)
  (declare (special *global*))
  (let ((path (make-pathname :name filename-as-string)))
    (with-open-file (str path :direction :input)
      (loop for line = (read str nil 'eof)
            until (eql line 'eof)
            do (normalize (internalize line) *global* (lambda (result!) result!))))))

;;; externalization.lsp

(in-package 3-proto-lisp)

;; Externalization

(defmethod external-type ((numeral numeral))
  'numeral)
```

```
(defmethod external-type ((number number))
  'number)

(defmethod external-type ((cl-boolean cl-boolean))
  'truth-value)

(defmethod external-type ((closure closure))
  'closure)

(defmethod external-type ((function function))
  'function)

(defmethod external-type ((rail rail))
  'rail)

(defmethod external-type ((wrapped-cl-list wrapped-cl-list))
  'sequence)

(defmethod external-type ((atom atom))
  'atom)

(defmethod external-type ((symbol symbol))
  'symbol)

(defmethod external-type ((pair pair))
  'pair)

(defmethod external-type ((cons cons))
  'procedure-call)

(defmethod external-type ((handle handle))
  'handle)

(defmethod print-to-string ((handle handle))
  (if (eql (class-of handle) (find-class 'handle))
      (format nil "?~a" (print-to-string (cl-value handle)))
    (format nil "~s" (cl-value handle))))

(defmethod print-to-string ((boolean boolean))
  (cond ((eql (unwrap boolean) *cl-true*) "$T")
        ((eql (unwrap boolean) *cl-false*) "$F")
        (t (error "print-to-string: Trying to print erronous boolean."))))

(defmethod print-to-string ((wrapped-cl-list wrapped-cl-list))
  (format nil "~s" (cl-list wrapped-cl-list)))

(defmethod print-to-string ((rail rail))
  (with-output-to-string (s)
    (format s "[ ")
    (loop for handle in (cl-list (unwrap rail))
          do (format s (print-to-string handle)))
    (format s " ]")))

(defmethod print-to-string (smth)
  (format nil "~a" smth))

(defmethod print-to-string ((primitive-closure primitive-closure))
  "<primitive procedure>")


(defmethod print-to-string ((reflective-closure reflective-closure))
  "<reflective procedure>")

(defmethod print-to-string ((closure closure))
  "<simple procedure>")
```

```
;;; internalization.lsp

(in-package 3-proto-lisp)

;; Added syntax

;; Rails
(set-macro-character
 #\] #'(lambda (stream char)
         (declare (ignore char))
         (read stream t nil t) nil))

(set-macro-character
 #\[ #'(lambda (stream char)
         (declare (ignore char))
         (make-instance 'wrapped-cl-list :cl-list (read-delimited-list #\] stream t))))

;; Booleans

(defmethod identify-cl-boolean ((symbol (eql 'T)))
  *cl-true*)

(defmethod identify-cl-boolean ((symbol (eql 'F)))
  *cl-false*)

(defmethod identify-cl-boolean (smth)
  (error "Error while parsing ~s, $ is reserved syntax for booleans." smth))

(set-macro-character
 #\$ #'(lambda (stream char)
         (declare (ignore char))
         (identify-cl-boolean (read stream t nil t))))

;; Handle (cf ')
;; In order not to confuse Common Lisp, we use ^ instead of ' here.
(set-macro-character
 #\^ #'(lambda (stream char)
         (declare (ignore char))
         (internalize (read stream t nil t))))

;; Internalize wraps CL values to internal structures
(defmethod internalize ((handle handle))
  (make-instance 'handle :cl-value handle))

(defmethod internalize ((number number))
  (make-instance 'numeral :cl-value number))

(defmethod internalize ((cl-boolean cl-boolean))
  (make-instance 'boolean :cl-value cl-boolean))

(defmethod internalize ((wrapped-cl-list wrapped-cl-list))
  (make-instance 'rail :cl-value (make-instance 'wrapped-cl-list
                                                :cl-list (cl-list wrapped-cl-list))))

(defmethod internalize ((symbol symbol))
  (make-instance 'atom :cl-value symbol))

(defmethod internalize ((cons cons))
  (make-instance 'pair :cl-value cons))
```

```
;;; normalization.lsp

(in-package 3-proto-lisp)

;; Normalization

;; Environments

(defclass environment (handle)
  ((bindings :initarg :bindings :initform '() :accessor bindings)))

(defmethod environment-p ((environment environment))
  T)

(defmethod environment-p (smth)
  nil)

(defmethod ccons ((atom atom) (environment environment) (rail rail) pair)
  (case (unwrap atom)
    (simple
     (make-instance 'closure
      :name 'anonymous :lexical-environment environment :argument-pattern rail :body pair))
    (reflect
     (make-instance 'reflective-closure
      :name 'anonymous :lexical-environment environment :argument-pattern rail :body pair))
    (t
     (error "ccons: unknown procedure type: ~s" (unwrap atom)))))

(defmethod print-to-string ((environment environment))
  "<environment>")

(defmethod make-mapping ((atom atom) value)
  (list atom value 'mapping))

(defmethod mapping-value (mapping)
  (if (and (listp mapping) (eql (car (last mapping)) 'mapping))
      (second mapping)
    (error "Mapping expected.")))

(defmethod set-binding ((environment environment) (atom atom) (handle handle))
  (push (make-mapping atom handle) (bindings environment)))

(defmethod add-binding ((environment environment) (atom atom) (handle handle))
  (make-instance 'environment
               :bindings (cons (make-mapping atom handle) (bindings environment))))

(defmethod bind ((environment environment) (argument-pattern rail) (arguments rail))
  (cond ((empty-p argument-pattern) environment)
        ((= (length argument-pattern) (length arguments))
         (bind (add-binding environment (first argument-pattern) (first arguments))
               (rest argument-pattern) (rest arguments)))
        (t
         (error "bind: Function called with the wrong number of arguments."))))

(defmethod binding ((atom atom) (environment environment))
  (let ((mapping (find (unwrap atom) (bindings environment)
                       :key (lambda (atom+value) (unwrap (car atom+value))))))
    (if (null mapping)
        (error "binding: variable ~s unbound in env." (unwrap atom))
      (mapping-value mapping))))

;; Global environment

(defparameter *global* (make-instance 'environment))
```

```
;; Normalize

(defun normalize (internal-structure environment continuation)
  (cond ((normal-p internal-structure)
         (funcall continuation internal-structure))
        ((atom-p internal-structure)
         (funcall continuation (binding internal-structure environment)))
        ((rail-p internal-structure)
         (normalize-rail internal-structure environment continuation))
        ((pair-p internal-structure)
         (reduce (pcar internal-structure) (pcdr internal-structure)
                 environment continuation))
        (t
         (error "normalize: Error trying to normalize non internal structure ~s"
                internal-structure))))

(defun normal-p (internal-structure)
  (if (rail-p internal-structure)
      (normal-rail-p internal-structure)
    (and (not (atom-p internal-structure))
         (not (pair-p internal-structure)))))

(defmethod normal-rail-p ((rail rail))
  (or (empty-p rail)
      (and (normal-p (first rail))
           (normal-rail-p (rest rail)))))

(defun normalize-rail (rail environment continuation)
  (if (empty-p rail)
      (funcall continuation (rcons))
    (normalize (first rail)
               environment
               (lambda (first!)
                 (normalize-rail (rest rail)
                                 environment
                                 (lambda (rest!)
                                   (funcall continuation (prep first! rest!))))))))

(defun reduce (procedure arguments environment continuation)
  (normalize procedure
             environment
             (lambda (procedure!)
               (cond ((reflective-p procedure!)
                      (reduce-reflective procedure! arguments environment continuation))
                     ((abnormal-p procedure!)
                      (reduce-abnormal procedure! arguments environment continuation))
                     (t
                      (normalize arguments
                                 environment
                                 (lambda (arguments!)
                                   (if (primitive-p procedure!)
                                       (funcall continuation
                                                (wrap (apply (unwrap procedure!)
                                                             (cl-list (unwrap arguments!)))))
                                   (normalize (body procedure!)
                                              (bind (lexical-environment procedure!)
                                                    (argument-pattern procedure!) arguments!)
                                              continuation))))))))

(defun reduce-reflective (procedure! arguments environment continuation)
 (let ((non-reflective-closure (de-reflect procedure!)))
  (normalize (body non-reflective-closure)
             (bind (lexical-environment non-reflective-closure)
                   (argument-pattern non-reflective-closure)
                   (wrap (make-instance 'wrapped-cl-list
                                        :cl-list (list environment continuation arguments))))
             (create-meta-continuation))))
```

```
(defun reduce-abnormal (procedure! arguments environment continuation)
  (ecase (name procedure!)
    (set (reduce-set arguments environment continuation))
    (lambda (reduce-lambda 'closure arguments environment continuation))
    (lambda-reflect (reduce-lambda 'reflective-closure arguments environment continuation))
    (if (reduce-if arguments environment continuation))
    (apply (reduce-apply arguments environment continuation))
    (apply-abnormal (reduce-apply-abnormal arguments environment continuation))))

(defun reduce-set (arguments environment continuation)
  (declare (special *global*))
  (let ((atom (first arguments)) ;; do not normalize the symbol
        (expression (first (rest arguments)))) ;; normalize the expression
    (normalize expression environment
               (lambda (expression!)
                 (set-binding *global* atom expression!)
                 (funcall continuation (wrap 'ok))))))

(defun reduce-if (arguments environment continuation)
  (let ((condition (first arguments))
        (consequent (first (rest arguments)))
        (antesequent (first (rest (rest arguments)))))
    (normalize condition environment
               (lambda (condition!)
                 (if (cl-bool (unwrap condition!)) ;; Not Lisp-style bools
                     (normalize consequent
                                environment
                                (lambda (consequent!)
                                  (funcall continuation consequent!)))
                   (normalize antesequent
                              environment
                              (lambda (antesequent!)
                                (funcall continuation antesequent!))))))))

(defun reduce-lambda (closure-class-name arguments environment
continuation)
  (let ((argument-pattern (first arguments))
        (body (first (rest arguments))))
    (funcall continuation
             (make-instance closure-class-name
                            :name 'anonymous
                            :body body
                            :argument-pattern
                            (wrap (make-instance 'wrapped-cl-list
                                                 :cl-list (unwrap argument-pattern)))
                            :lexical-environment environment))))

(defmethod de-reflect ((reflective-closure reflective-closure))
  (make-instance 'closure
                 :name (name reflective-closure)
                 :argument-pattern (argument-pattern reflective-closure)
                 :body (body reflective-closure)
                 :lexical-environment (lexical-environment reflective-closure)))

(defun reduce-apply (arguments environment continuation) ;; apply takes a symbol and a rail
  (normalize arguments environment
             (lambda (arguments!)
               (reduce (first arguments!) (unwrap (first (rest arguments!)))
                       environment continuation))))
```

```
(defun reduce-apply-abnormal (arguments environment continuation)
  (normalize arguments environment
             (lambda (arguments!)
               (reduce (first arguments!)
                       (unwrap (first (rest (rest arguments!))))
                       (unwrap (first (rest arguments!)))
                       continuation))))


;;; structural-field.lsp

(in-package 3-proto-lisp)

;; Structural field

;; Classes implementing the ProtoLisp types.
;; Instances of these classes are the internal representation of ProtoLisp values.
;; Instances of these classes are called internal structures.

(defclass handle ()
  ((cl-value :initarg :cl-value :accessor cl-value)))

;; NUMBER

(defclass numeral (handle)
  ())

;; BOOLEAN

(defclass boolean (handle)
  ())

;; CLOSURE

(defclass closure (handle)
  ((body :initarg :body :accessor body :initform nil)
   (lexical-envrionment :initarg :lexical-environment :accessor lexical-environment :initform nil)
   (name :initarg :name :accessor name :initform nil) ; for debugging
   (argument-pattern :initarg :argument-pattern :accessor argument-pattern :initform nil)))

(defclass primitive-closure (closure)
  ())

(defclass reflective-closure (closure)
  ())

;; Abnormal closures are primitive closures whose arguments are not all normalized

(defclass abnormal-closure (primitive-closure)
  ())

;; RAIL

(defclass rail (handle)
  ())

;; Helper class, for representing rails in CL

(defclass wrapped-cl-list ()
  ((cl-list :initarg :cl-list :accessor cl-list)))

;; ATOM

(defclass atom (handle)
  ())

(defclass pair (handle)
  ())
```

```
;; Operations boolean

(defmethod boolean-p ((boolean boolean))
  T)

(defmethod boolean-p (smth)
  nil)

;; Operations atom

(defmethod atom-p ((atom atom))
  T)

(defmethod atom-p (smth)
  nil)

;; Operations numeral

(defmethod numeral-p ((numeral numeral))
  T)

(defmethod numeral-p (smth)
  nil)

;; Operations pair

(defmethod pair-p ((pair pair))
  T)

(defmethod pair-p (smth)
  nil)

;; Operations closure

(defmethod primitive-p ((primitive-closure primitive-closure))
  T)

(defmethod primitive-p (smth)
  nil)

(defmethod reflective-p ((reflective-closure reflective-closure))
  T)

(defmethod reflective-p (smth)
  nil)

(defmethod abnormal-p ((abnormal-closure abnormal-closure))
  T)

(defmethod abnormal-p (smth)
  nil)

(defmethod closure-p ((closure closure))
  T)

(defmethod closure-p (smth)
  nil)

;; Operations pair

(defmethod pcar ((pair pair))
  (wrap (car (unwrap pair))))

(defmethod pcdr ((pair pair))
  ;; returns a rail
  (wrap (make-instance 'wrapped-cl-list :cl-list (cdr (unwrap pair)))))
```

```
(defmethod pcons ((handle1 handle) (handle2 handle))
  (wrap (cons (unwrap handle1) (unwrap handle2))))

;; Operations rail

(defmethod rail-p ((rail rail))
  T)


(defmethod rail-p (smth)
  nil)

(defmethod rcons (&rest list-of-structures)
  (wrap (make-instance 'wrapped-cl-list :cl-list (mapcar (function unwrap) list-of-structures))))

(defmethod scons (&rest list-of-structures)
  (make-instance 'wrapped-cl-list :cl-list list-of-structures))

(defmethod pcons ((handle handle) (rail rail))
  (wrap (cons (unwrap handle) (cl-list (unwrap rail)))))

(defmethod prep ((handle handle) (rail rail))
  (wrap (make-instance 'wrapped-cl-list :cl-list (cons (unwrap handle) (cl-list (unwrap rail))))))

(defmethod prep (smth (wrapped-cl-list wrapped-cl-list))
  (make-instance 'wrapped-cl-list :cl-list (cons smth (cl-list wrapped-cl-list))))

(defmethod length ((rail rail))
  (cl:length (cl-list (unwrap rail))))

(defmethod nth (nr (rail rail))
  (wrap (cl:nth nr (cl-list (unwrap rail)))))

(defmethod nth (nr (wrapped-cl-list wrapped-cl-list))
  (cl:nth nr (cl-list wrapped-cl-list)))

(defmethod pcar ((rail rail))
  (nth 0 rail))

(defmethod pcdr ((rail rail))
  (tail 1 rail))

(defmethod tail (nr (rail rail))
  (wrap (make-instance 'wrapped-cl-list :cl-list (cl:nthcdr nr (cl-list (unwrap rail))))))

(defmethod tail (nr (wrapped-cl-list wrapped-cl-list))
  (make-instance 'wrapped-cl-list :cl-list (cl:nthcdr nr (cl-list wrapped-cl-list))))

(defmethod first ((rail rail))
  (wrap (cl:first (cl-list (unwrap rail)))))

(defmethod rest ((rail rail))
  (wrap (make-instance 'wrapped-cl-list :cl-list (cl:rest (cl-list (unwrap rail))))))

(defmethod rest ((wrapped-cl-list wrapped-cl-list))
  (make-instance 'wrapped-cl-list :cl-list (cl:rest (cl-list wrapped-cl-list))))

(defmethod empty-p ((rail rail))
  (null (cl-list (unwrap rail))))

(defmethod empty-p ((wrapped-cl-list wrapped-cl-list))
  (null (cl-list wrapped-cl-list)))

;; Equality

(defmethod proto-lisp= ((handle1 handle) (handle2 handle))
  (proto-lisp= (unwrap handle1) (unwrap handle2)))
```

```
(defmethod proto-lisp= ((rail1 rail) (rail2 rail))
  (declare (special *cl-false*))
  *cl-false*)

(defmethod proto-lisp= ((handle1 handle) smth)
  (declare (special *cl-false*))
  *cl-false*)


(defmethod proto-lisp= (smth (handle1 handle))
  (declare (special *cl-false*))
  *cl-false*)

(defmethod proto-lisp= (smth1 smth2)
  (declare (special *cl-true* *cl-false*))
  (if (equal smth1 smth2)
      *cl-true*
    *cl-false*))

;; Internalization & Parsing

;; Reads from the standard input, these are plain CL values, which are mapped onto ProtoLisp values.
;; Most syntax of ProtoLisp overlaps with CL's syntax, for other cases there is a read macro.


(defun three-lisp-read-and-parse ()
  (let ((input (read)))
    (internalize input)))

;; Helper classes because syntax per type is unique in ProtoLisp, but not in CL.

(defclass cl-boolean ()
  ())

(defparameter *cl-true* (make-instance 'cl-boolean))
(defparameter *cl-false* (make-instance 'cl-boolean))

(defmethod cl-bool ((obj (eql *cl-true*)))
  T)

(defmethod cl-bool ((obj (eql *cl-false*)))
  nil)

(defmethod cl-bool (smth)
  (error "cl-bool: ~s is not either *cl-true* or cl-false" smth))

(defun cl->cl-bool (smth)
  (if smth
      *cl-true*
    *cl-false*))

(defmethod proto-lisp= ((smth1 wrapped-cl-list) (smth2 wrapped-cl-list))
  (if (equal (cl-list smth1) (cl-list smth2))
      *cl-true*
    *cl-false*))

;; sequence equivalent
(defmethod length ((wrapped-cl-list wrapped-cl-list))
  (cl:length (cl-list wrapped-cl-list)))

;; Wrapping and Unwrapping

(defclass cl-closure ()
  ((3l-closure :initarg :closure :reader closure))
  (:metaclass funcallable-standard-class))

(defmethod cl-closure-p ((cl-closure cl-closure))
  T)
```

```
(defmethod cl-closure-p (smth)
  nil)

(defmethod wrap ((cl-closure cl-closure))
  (closure cl-closure))

(defmethod wrap (smth)
  (internalize smth))

(defmethod wrap ((function function))
  (make-instance 'primitive-closure :cl-value function))

(defmethod unwrap ((handle handle))
  (cl-value handle))

(defmethod unwrap ((primitive-closure primitive-closure))
  (cl-value primitive-closure))

(defmethod unwrap ((reflective-closure reflective-closure))
  (let ((cl-closure (make-instance 'cl-closure :closure reflective-closure)))
    (set-funcallable-instance-function
     cl-closure
     (lambda (&rest args)
       (declare (ignore args))
       (error "Don't call reflective closures within Common Lisp code.")))
    cl-closure))

(defmethod unwrap ((closure closure))
  (declare (special *global*))
  (let ((cl-closure (make-instance 'cl-closure :closure closure)))
    (set-funcallable-instance-function
     cl-closure
     (lambda (args)
       (reduce closure
              (make-instance 'rail
                             :cl-value (make-instance 'wrapped-cl-list :cl-list (list args)))
              *global*
              (create-meta-continuation))))
   cl-closure))

;;; primitives.lsp

(in-package 3-proto-lisp)

;; Primitives

(defun set-primitive (name lambda)
  (declare (special *global*))
  (set-binding *global* (wrap name) (wrap lambda)))

(defun set-primitive-abnormal (name lambda)
  (declare (special *global*))
  (declare (special *cl-false*))
  (set-binding *global* (wrap name)
               (make-instance 'abnormal-closure
                              :body lambda :name name :cl-value (wrap *cl-false*))))

;; arithmetic
(set-primitive '+ (function +))
(set-primitive '- (function -))
(set-primitive '* (function *))
(set-primitive '/ (function /))
(set-primitive '= (function proto-lisp=))
(set-primitive '< (lambda (cl-val1 cl-val2) (cl->cl-bool (< cl-val1 cl-val2))))
(set-primitive '> (lambda (cl-val1 cl-val2) (cl->cl-bool (> cl-val1 cl-val2))))
```

```
;; printing

(set-primitive 'print (function print))
;; pair
(set-primitive 'pcar (function pcar))
(set-primitive 'pcdr (function pcdr))
(set-primitive 'pcons (function pcons))
;; rails and sequences
(set-primitive 'rcons (function rcons))
(set-primitive 'scons (function scons))
(set-primitive 'prep  (function prep))
(set-primitive 'length (function length))
(set-primitive 'nth (function nth))
(set-primitive 'tail (function tail))
(set-primitive 'empty (lambda (rail) (cl->cl-bool (empty-p rail))))
;; closure
(set-primitive 'body (function body))
(set-primitive 'pattern (function argument-pattern))
(set-primitive 'environment (function lexical-environment))
(set-primitive 'ccons (function ccons))
;; atoms
(set-primitive 'acons (function gensym))
;; typing
(set-primitive 'type (function external-type))
(set-primitive 'primitive (lambda (closure) (cl->cl-bool (primitive-p closure))))
(set-primitive 'reflective (lambda (closure) (cl->cl-bool (reflective-p closure))))
;; up & down
(set-primitive 'up (function wrap))
(set-primitive 'down (function unwrap))
;; abnormal primitives, i.e. primitives whose args are not normalized
(set-primitive-abnormal
 'set
(lambda (&rest args) (declare (ignore args)) (error "Trying to call set")))
(set-primitive-abnormal
 'lambda
(lambda (&rest args) (declare (ignore args)) (error "Trying to call lambda")))
(set-primitive-abnormal
 'lambda-reflect
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call lambda-reflect")))
(set-primitive-abnormal
 'if
(lambda (&rest args) (declare (ignore args)) (error "Trying to call if")))

(set-primitive-abnormal
 'apply
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call apply")))
(set-primitive-abnormal
 'apply-abnormal
 (lambda (&rest args) (declare (ignore args)) (error "Trying to call apply-abnormal")))
;; global environment
(set-primitive 'global *global*)
(set-primitive 'binding (function binding))
(set-primitive 'bind (function bind))
;; normalize
(set-primitive 'normalize (function normalize))
```

Back to the Future in One Week — Implementing a Smalltalk VM in PyPy

Carl Friedrich Bolz[2], Adrian Kuhn[1], Adrian Lienhard[1], Nicholas D. Matsakis[3], Oscar Nierstrasz[1], Lukas Renggli[1], Armin Rigo, and Toon Verwaest[1]

[1] Software Composition Group
University of Bern, Switzerland
[2] Softwaretechnik und Programmiersprachen
Heinrich-Heine-Universität Düsseldorf
[3] ETH Zurich, Switzerland

Abstract. We report on our experiences with the SPY project, including implementation details and benchmark results. SPY is a re-implementation of the Squeak (*i.e.,* Smalltalk-80) VM using the PyPy toolchain. The PyPy project allows code written in RPython, a subset of Python, to be translated to a multitude of different backends and architectures. During the translation, many aspects of the implementation can be independently tuned, such as the garbage collection algorithm or threading implementation. In this way, a whole host of interpreters can be derived from one abstract interpreter definition. SPY aims to bring these benefits to Squeak, allowing for greater portability and, eventually, improved performance. The current SPY codebase is able to run a small set of benchmarks that demonstrate performance superior to many similar Smalltalk VMs, but which still run slower than in Squeak itself. SPY was built from scratch over the course of a week during a joint Squeak-PyPy Sprint in Bern last autumn.

1 Introduction

In this paper we present a preliminary report on the SPY project. SPY is an implementation of the Squeak [4] variant of Smalltalk built using the PyPy toolchain [8]. The goals of the SPY project are to allow the popular Squeak platform to be easily ported to high-level runtimes, such as the Java Virtual Machine (JVM) and Common Language Runtime (CLR), as well as to eventually improve Squeak's performance through the use of PyPy's Just-in-time (JIT) compiler generation techniques. The SPY project also serves to highlight some of the distinctive features in PyPy's approach to building virtual machines, especially when it is compared to Squeak.

Squeak is an open source, full-featured implementation of Smalltalk. One of its distinctive features is that the virtual machine itself is written in Slang [4], a restricted subset of Smalltalk. Slang is designed to be easily translated into C, meaning that the core VM can be translated into C and then compiled with a standard C compiler. This allows Squeak to enjoy reasonably fast execution and high portability, while preserving the ability to read and understand the VM source code without leaving Smalltalk.

The PyPy project[1] is a toolchain for building interpreters [8]. It allows interpreters to be written in RPython, a restricted form of Python, and then translated to a more efficient lower-level language for execution. PyPy is able to translate RPython programs to many different backends, ranging from C source code, to JavaScript (for execution in the browser), to bytecodes for the JVM or CLR, although not all of these backends are as full-featured as the others. In addition to simple translation, PyPy can introduce optimizations along the way, and can even generate a just-in-time compiler semi-automatically from the interpreter source. These features are described in depth in other publications [1,6].

At first glance, it may seem that the role of Slang in Squeak and RPython in SPY /PyPy are very similar. Both are restricted forms of a dynamic language used to code the core of the interpreter. However, the similarity is only skin deep. Slang, the restricted form of Smalltalk used by Squeak, is designed to be easily translated to C. Slang only contains constructs that can be directly mapped to C. RPython, on the other hand, is much closer to the full Python language and includes such features as garbage collection, classes with virtual functions, and exceptions. Having such facilities available frees the programmer to focus on the language design issues and ignore the mundane, low-level details of writing a VM.

The main contributions of this paper are

– We report on our experiences using the PyPy toolchain to realize SPY, a Smalltalk-80 VM.
– We present implementation details and discuss design decisions of the realized VM.
– We compare benchmarks of SPY with those of similar Smalltalk VMs.

The remainder of this paper is structured as follows: In Section 2 we present a brief overview of the PyPy project. In Section 3 we explain how the PyPy approach has been adapted in the SPY project to the implementation of a Squeak VM. Related work is presented in Section 4. Section 5 presents the results of various benchmarks to validate the effectiveness of the SPY implementation. We provide remarks on future work in Section 6 and eventually present in Section 7 our conclusions. Additionally, the source code of the benchmarks is given in Appendix A, and download and build instructions for both PyPy and SPY are given in Appendix B.

2 PyPy in a Nutshell

Although the initial goal of the PyPy project was to implement a next-generation interpreter for Python, the project has gradually evolved into a general-purpose tool that can be used for any number of languages. In addition to Python and Smalltalk, (partial) interpreters have been developed for Prolog, Scheme and JavaScript.

[1] `http://codespeak.net/pypy/`

The goal of the PyPy project is to create an environment that makes it easy to experiment with different virtual machine designs, but without sacrificing efficiency. This is achieved by separating the semantics of the language being implemented, such as Python or Smalltalk, from low-level aspects of its implementation, such as memory management or the threading model. A complete interpreter is constructed at build time by weaving together the interpreter definition and each low-level aspect into a complete and efficient whole [8].

The project currently includes a wide variety of backends that support translations from RPython into C/Posix, LLVM[5], CLI/.NET, Java bytecodes, and JavaScript, although only the first three are fully functional.

The translation process works by using abstract interpretation to convert the RPython programs into flow graph form. The graphs are then used for whole-program type inference, which assigns a static type to all values used in the program. The ability to perform type inference on the input programs is the key requirement for the PyPy toolchain. This means that RPython is defined, rather imprecisely, to be the subset of Python which our tools can statically check. In practice, RPython forbids runtime reflection and any type-inconsistent usage of variables (*e.g.,* assigning both integers and object references to the same variables). Despite these restrictions, RPython is still quite expressive, supporting exceptions, single inheritance with explicitly declared mixins (instead

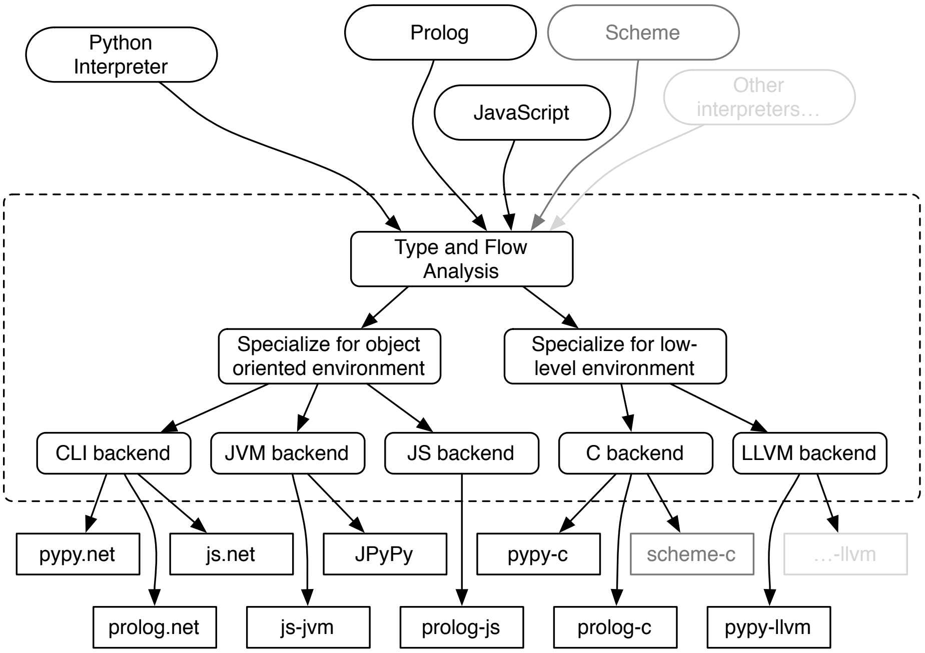


Fig. 1. The PyPy toolchain specializes high-level interpreters for different languages into compilers or VMs for different platforms

of Python's full multiple inheritance), dynamic dispatch, first class function and class values, and runtime `isinstance` and type checks.

Once the flow graphs have been built and typed, they can be transformed by any number of translation aspects[1] which implement low-level details, such as garbage collection or a variant of the CPS-transformation. These translation aspects give tremendous flexibility in controlling the behavior and performance of the final interpreter and also illustrate one of the advantages of specifying the interpreter in a higher-level language like RPython. Because RPython does not specify low-level details such as the garbage collection strategy, the toolchain is free to implement them however it sees fit. In contrast, changing the traditional, C-based Python interpreter so as not to use reference counting would require pervasive changes throughout the entire codebase.

The promise of PyPy and RPython is that it should be possible to develop a single interpreter source which can be used via different choices of translation aspects and backends, to create a whole family of interpreters on a wide variety of platforms, as illustrated by Figure 1. This avoids the problem that many languages face, *i.e.,* to keep the interpreter definition in sync across all platforms on which it is supported, and to allow all versions to benefit from new features and optimizations instantly. As an example, consider the Jython project[2], which defines a Python interpreter on the JVM. Because Jython and CPython do not share the same source, Jython lags several versions behind its C counterpart, making it increasingly challenging to use with modern Python programs. PyPy essentially offers a model-driven approach [9] to programming language implementation — it transforms platform-independent models (*i.e.,* high-level interpreters) into implementations for multiple platforms.

Another advantage of this approach is that since RPython is a proper subset of Python, an RPython program can be fully tested and easily debugged by letting it run on a Python interpreter. The program is only translated to a different language if it is reasonably bug-free. This increases testability, eases development, and decreases turnaround times.

3 Spy Implementation

Similar to most Smalltalk VMs, Spy consists of four main parts: a bytecode interpreter, a set of primitives, an image loader, and an object model.

3.1 Interpreter, Primitives, and Image Loading

The core components of a Smalltalk VM are the bytecode interpreter, primitive methods, and the image loader. For the most part Spy does not deviate from the traditional Smalltalk VM design [3], though in some cases we made minor alterations. For example, Spy is not based on an object table, *i.e.,* objects reference each other directly without a level of indirection. This is similar to Squeak's approach as described in Section 4.

[2] http://jython.org/

```
Before:
table = [method_for_opcode_0, method_for_opcode_1, ...]
while 1:
    byte = get_next_byte()
    method = table[byte]
    method()

After:
while 1:
    byte = get_next_byte()
    switch on byte:
        case 0:
            method_for_opcode_0()
        case 1:
            method_for_opcode_1()
        ...
```

Fig. 2. Translation of the dispatch loop from a bytecode table to a local switch

Bytecodes in Smalltalk are generally used to implement control flow and message sends, and to introduce constant values into the computation. SPY's bytecode interpreter takes a traditional form, consisting of a table of function pointers, which is indexed by the current bytecode on every iteration.

As a performance optimization, during the translation process to C, we are able to take advantage of the fact that the function table is immutable. This allows us to alter the dispatch loop so that it uses a local `switch` to translate bytecodes to method calls, rather than having an indirection via the global opcode table (see Figure 2). This will not only localize lookups but it will also use direct instead of indirect calls. Which will then allow for further optimizations such as inlining of the actual code related to the bytecodes.

Compared to other virtual machines, Smalltalk contains relatively few bytecodes. For example, there are no bytecodes for low-level operations such as doing arithmetic. Instead, these operations are implemented as *primitive methods*, which are methods that are implemented in the core virtual machine, either for efficiency's sake or because they encode a fundamental operation which isn't possible to express in the language itself, such as integer addition.

Primitive methods are invoked as the result of normal message sends. When an object receives a message in Smalltalk, the first thing that happens is the lookup of the corresponding method implementation. The resulting method object contains the bytecodes to execute and, optionally, a primitive method identifier, which is just a small integer. If a primitive method identifier is supplied, the VM uses the integer to index into its primitive method table to find a built-in function to execute.

As in Squeak, primitive methods in SPY are implemented as a series of functions placed into a table. In SPY, however, we are able to take advantage of several RPython features to make their implementation less tedious and error-prone. The first feature are exceptions: in the Squeak VM, when a primitive

```
@expose_primitive(FLOAT_SQUARE_ROOT, unwrap_spec=[float])
def func(interp, f):
    if f < 0.0:
        raise PrimitiveFailedError
    w_res = utility.wrap_float(math.sqrt(f))
    return w_res
```

Fig. 3. The definition of the primitive square root operation in RPython. The code uses high-level features, such as method decorators, exceptions, and object-orientation.

method wants to signal failure, it does so by setting a field, `primitiveFailed`, of the global interpreter object to true. This means that all following code must be guarded to ensure that it does not execute once the `primitiveFailed` field is set to true. In RPython, however, we can use a Python exception to signal a failure condition, resulting in less cluttered code.

The second RPython feature, which proved to be very important is its capacity for meta-programming. Because primitive methods execute directly on the VM structures, they often contain a certain amount of repetitive code that loads method arguments from the stack, inspects their types, and finally pushes any result back onto the stack. Using Python annotations, however, we are able to attach a declarative tag to each primitive method stating the number of stack arguments it expects, any preprocessing they require, and whether or not a result is pushed back on the stack after execution. This not only makes the primitives shorter, it helps to avoid errors. In particular, we were able to use these annotations to specify when an argument represents an array index: since array indices are 1-based in Smalltalk, the preprocessor is not only able to confirm that the index is an integer, but can automatically subtract 1 to convert it to a 0-based RPython array index, leading to much cleaner code.

Figure 3 shows the definition of the primitive method for computing square roots. The `@expose_primitive` annotation on the first line declares both the primitive code, which is the symbolic constant `FLOAT_SQUARE_ROOT`, and the fact that the function expects only one argument from the stack, which should be a floating point value. Note that the object on the stack is actually a wrapped floating point value, but the preprocessor automatically inserts code to unwrap it and extract the RPython floating point value within. This unwrapped value is passed to the implementation function. Within the body of the function, there is a test that ensures that the argument is positive which raises an exception (`PrimitiveFailedError`) should that not be the case. Otherwise, the square root is computed using the standard RPython function `math.sqrt`, wrapped in a Smalltalk object, and returned. Note that the return value will be automatically pushed on the stack.

For comparison, Figure 4 shows the the same primitive method in Slang. The key difference to RPython is that Slang does not provide object-oriented language features. Slang is, roughly spoken, C code disguised as Smalltalk syntax.

For example, to indicate failure, a global field of the interpreter is used rather than throwing an exception. Pushing and popping has to be done manually.

```
primitiveSquareRoot
    | rcvr |
    self var: #rcvr type: 'double '.
    rcvr := self popFloat.
    self success: rcvr >= 0.0.
    successFlag
        ifTrue: [self pushFloat:
            (self cCode: 'sqrt(rcvr)' inSmalltalk: [rcvr sqrt])]
        ifFalse: [self unPop: 1]
```

Fig. 4. The definition of the primitive square root operation in Slang. The code is, roughly spoken, C code disguised as Smalltalk syntax. Object-oriented features are not used, *e.g.,* failure is signalled with flag rather than an exception.

But in particular the call to `#cCode:inSmalltalk:` breaks abstractions and testability: as a first argument it is given a fragment of C code, as a second argument a Smalltalk closure. When translating the VM down to C, the code fragment is literally copied into the generated source code. When debugging the VM within another Smalltalk image, the closure is evaluated. As both are not causally connected, it might even happen that a bug in the C code does not appear when debugging the VM and vice versa!

Image loading is one area where SPY differs significantly from Squeak. Traditionally, a Squeak image is simply a dump of the core memory into a file. Loading an image can be done by simply memory-mapping the image file, followed by some minimal pointer and integer adjustments. This technique works well when you can guarantee that the memory layout between virtual machines is compatible. Unfortunately, the memory layout for a RPython program is not specified and sometimes even outside of the control of PyPy's toolchain, if the translation target is a high-level VM such as the JVM or .NET's CLR. Since we wanted to retain compatibility with Squeak's image formats, we implemented an image loader that proceeds by parsing the Squeak image file formats, decoding the object headers, and constructing equivalent objects in our own VM.

3.2 Object Model

The Squeak implementation uses three different addressing schemes for its objects: bytes, words, and pointers. Each object structure begins with a format word that describes which kind of object it is. This determines how the raw bytes that make up an object in memory are interpreted: a "bytes" object treats them as an array of bytes, which is useful for classes like strings. "Words" objects store words, and are useful for vectors of integers. Finally, "pointers" objects contain pointers, and are used for almost all other kinds of objects. In addition, as is common in many VMs, small integers are represented as tagged pointers.

The SPY model is somewhat more complex. In addition to bytes, words, and pointers objects, we have special classes for representing compiled methods, method and block contexts (stack frames), small integers, and floating point values. Please refer to Figure 5 for the full class hierarchy.

All these classes are subclasses of an abstract class representing a Smalltalk object (`W_Object`). Therefore they all implement the same interface, which makes them equivalent from the Smalltalk point of view, and any of them can be used anyplace that a standard Smalltalk object is expected. The concrete implementation, however, differs from class to class. For the classes that are close to the VM internals (such as compiled methods, method and block contexts), the implementation is made as convenient for the VM as possible, whereas the implementation of generic Smalltalk objects is less complex.

As illustrated by Figure 5, small integers are implemented by a normal class that has exactly one field, which contains the value of the integer itself. This deviates from the original design in the Squeak VM where they are represented by tagged pointers. To compare results of both styles, we could easily mimic the behavior of the Squeak VM by plugging an extra transformation into the toolchain. With the transformation turned on, the resulting C source generated by the toolchain would actually use tagged pointers as representation of small integers. This is by itself already another example where the RPython code abstracts over low-level details. We can assume a consistent model everywhere and do not need to check for tagged pointers throughout the source code, while resulting in the exact same behavior. A small bit of experimentation seemed to indicate that using tagged pointers for small integers actually worsens performance. The necessity of checking whether a pointer is a heap pointer or a small integer around every method call offsets all the benefits of the smaller memory footprint that comes with tagged small integers. It is important to note, however, how tedious it would be to experimentally introduce or remove tagged pointers with a traditional, low-level interpreter.

The class hierarchy illustrated by Figure 5 is internal to the VM, it is not related to Squeak's class hierarchy. All these classes are internally used for

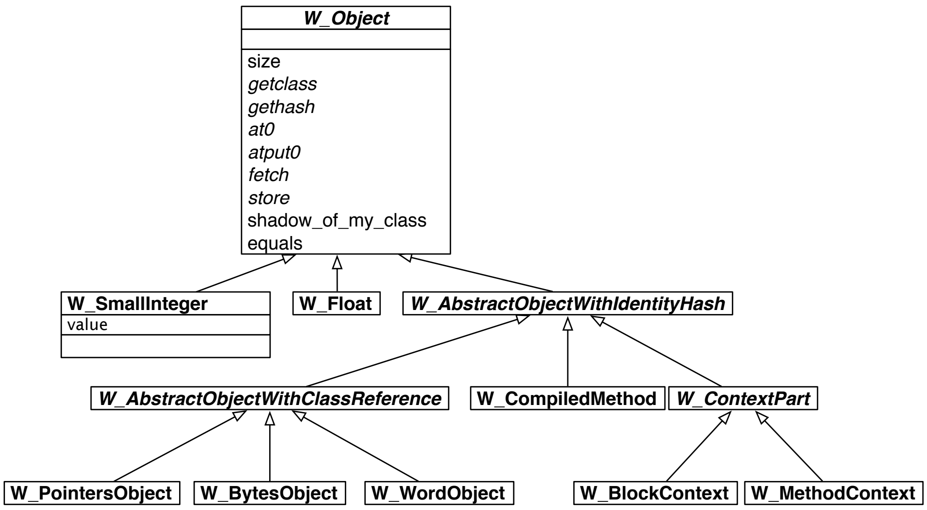


Fig. 5. Different kinds of objects in the Spy implementation

wrapper objects, hence the `W` prefix, and do not denote the high-level class of objects. Which high-level class an object has is completely under control of Squeak itself, it is stored in the `W_Object.shadow_of_my_class` field. Thus, the VM's class hierarchy can be used to run any version of Squeak. Both Squeak 2.0 and Squeak 3.9 images run with the current implementation of SPY.

3.3 Shadow Objects

As noted in the previous section, Squeak does not distinguish between objects based on the role that they play in the system, but only based on the kind of data that they contain (bytes, words, or objects). For example, a class object is simply an object that is *used as a class* by some other object. It is not necessarily an instance of a particular class, though the layout of the object in memory must be compatible with what the VM expects[3]. This implies that, at image loading time, it is impossible to distinguish which objects are, or will be later, used as classes, and so we cannot use a special subclass of `W_Object` to represent them.

Unfortunately, being forced to use a generic data layout for such special kinds of objects as classes can be very inefficient. The memory layout of Smalltalk data structures were chosen with an eye towards reducing memory consumption, and not for ease of access. SPY could be made far more efficient if it could use native RPython data structures instead. For example, each class has a method dictionary that is normally stored as a native Smalltalk `Dictionary` instance. If this method dictionary could be converted by the VM into a native RPython dictionary, then SPY could take advantage of the highly optimized RPython dictionary implementation.

To resolve this dilemma, SPY allows every Smalltalk object to have an associated "shadow" object. These shadow objects are not exposed to the Smalltalk world. They are used by the VM as internal representation and can hold arbitrary information about the actual object. If an object has a shadow object attached, the shadow is notified whenever the state of the actual object changes, to keep both views of the object synchronized. One way of looking at shadows is as a general cache mechanism. However, the approach is far more powerful, since arbitrary meta-level operations can be triggered when the update notifications are received[4].

In the current implementation, the shadows are used to attach nicely decoded information about classes to all objects which are used as classes. This allows any object to be used as a class, even ones that are not instances of the

[3] To be used as a class, a Squeak object must have at least three instance variables, of which the first must refer to its superclass, the second must refer to a method dictionary, and the third must contain a magic number encoding the format of instances. Any object that meets these criteria, and implements primitive #70 (primitiveNew), can be used to create instances of itself.

[4] Shadow objects are not related to the concept of mirrors. Mirrors are a mechanism to introduce reflection on demand. Shadow objects are an implementation artifact of our design allowing us to benefit from native RPython data structures. They have nothing to do with reflection per se, though they could be used for this purpose too.

Smalltalk class `Class`. The shadow object is created and attached to the class the first time the VM sees the object being used as a class. It stores all required information about the class in a convenient, easily accessible data structure (as opposed to the obscure bit format used at the Smalltalk level). The class shadow contains the format and size of instances, an RPython-level dictionary containing the compiled methods, and the name of the class (if it has one). All of this information is kept in sync with the "real" Smalltalk object that the shadow is associated with.

At the moment[5] classes are the only objects that have shadows attached to them. It seems likely that we will change some of the objects that are now implemented with special RPython classes to use shadows as well later. For example, Squeak allows arbitrary objects to be used as method and block contexts, but the current SPy implementation does not. This could be resolved by using shadow objects to contain any extra information associated with objects that are used as a context.

3.4 State of the Implementation

The VM parts described in the previous subsections add up to the current implementation of SPy. This implementation is able to load Squeak images (tested with the Squeak 2.0 mini image and more recent Squeak 3.9 images) and execute all bytecodes and a subset of primitives. The most important missing primitives are the graphical primitives. We do already support enough for the VM to be able to run the `tinyBenchmarks`. Furthermore we are still lacking support for threading and image saving.

4 Related Work

In this section we present existing Smalltalk VMs, and discuss how their implementation relates to SPy. The VM of the Squeak dialect of Smalltalk follows closely the specification given in the Smalltalk-80 Blue Book[3]. The Blue Book specifies an object memory format, the bytecodes, the primitives, and the interpreter loop of a Smalltalk VM.

4.1 Squeak VM

The main difference of Squeak's VM [4] compared to the Smalltalk-80 specification of the Blue Book [3] is the object memory format. The object memory specified in the blue book is based on an object table. An object table introduces a level of indirection for object references. In contrast, Squeak implements object references as direct pointers, that is, an object reference is just the address of that object in memory. Today, this is the common approach taken by most virtual machines.

[5] This paper refers to revision 49630 of SPy on codespeak's SVN repository, for more information please refer to the download instructions in Appendix B.

Squeak's object memory layout consists of a header for the class pointer, hash bits, GC flags, size *etc.* and a fixed number of fields. There are four kinds of object formats. Objects with named instance variables, indexed object fields, indexed word size or byte size fields. Everything, including interpreter-internal data such as execution contexts, processes, classes, and methods, is represented as a normal object on the heap. An exception is the case of small integers, which are represented as tagged pointers. Special objects that have to be known to the interpreter, for example the process scheduler, are stored in a global table.

The majority of the Squeak VM is implemented in a subset of Squeak Smalltalk, named Slang. The Slang source code is then translated to C code to compile and link with the low-level, platform-specific C code. Slang is a very restricted subset of Smalltalk which does not support classes, exception handling, or other object-oriented language features. Therefore, Slang does not provide a higher level of abstraction than C.

Nevertheless, using Slang has advantages compared to writing C code manually. First, the translator applies several optimizations such as generating C switch statements for the dispatch loop or inlining procedure calls. Second, since Slang is a Smalltalk subset it can be run within another Squeak image, which can be very useful for debugging. As Squeak allows for incremental compilation, the implementation of the VM can be changed while it is running. In this way, time consuming edit-transform-compile-run cycles can be avoided.

The approach taken by PyPy is similar to that of Squeak/Slang, as the VM implemented in RPython can also be run directly without transformation and compilation. However, the key difference is that RPython (restricted Python) is much less restrictive than Slang. RPython provides object-oriented language features such as objects, class-based inheritance, exceptions, and translation-time reflection and metaprogramming [8,7]. As discussed throughout this paper, RPython's extended capabilities simplify the implementation of the VM in many ways, ranging from using code generation and annotations to avoid boilerplate code, to the automation of complex, far-reaching optimizations like tagged integers.

5 Evaluation

In this section we present a comparison of performance and codebase size of different Smalltalk VM implementations.

5.1 Performance Benchmarks

To analyze VM performance, we use the TinyBenchmarks suite which is part of the Squeak mini image [2]. The TinyBenchmarks tests bytecode interpretation and message send performance. We refer the reader to Appendix A for the complete source code of the benchmark suite. For the Smalltalk platforms that do not support direct loading of our reference image, we ported the source code manually. All the platforms successfully run the TinyBenchmarks and produced the following two figures:

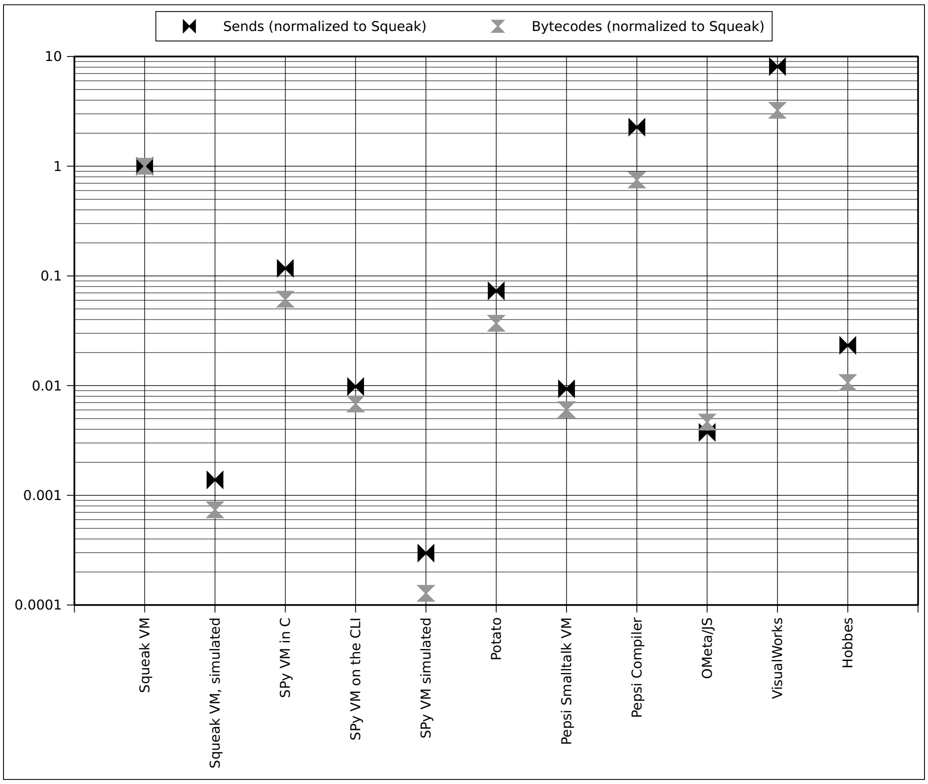


Fig. 6. Benchmark results for the TinyBenchmarks for various VMs, normalized to the Squeak VM

Bytecodes per second. To compare the performance of a virtual machine, we need to know how fast the bytecodes are processed by the VM. This is the first number reported by TinyBenchmarks. The value is calculated from a bytecode-heavy implementation of the "Sieve of Eratosthenes". The result is calculated using the runtime performance of this algorithm and the number of executed bytecodes. The number of executed bytecodes is the number of bytecodes that Squeak executes when running the benchmark, which makes the number meaningful even on different implementations.

Sends per second. In Smalltalk everything happens by message sends, with the exception of some transparently inlined control structures. Therefore an efficient implementation of message sends is crucial. The second number reported by TinyBenchmarks is the message sends (method lookup and method invocation) per second. It is calculated from the the runtime performance of a send-heavy recursive calculation of Fibonacci numbers.

In Figure 6 we present the result of running the TinyBenchmarks on various VMs in relation to the original Squeak VM. The machine used was an Apple Mac

Pro (2 × 3 GHz Dual-Core Intel Xeon, 3 GB RAM). All the benchmarks were run 20 times; the final numbers are the arithmetic mean of those measurements.

Squeak VM. This is the original Squeak VM, written in Slang and transformed to C. It is heavily optimized and represents our point of comparison.

Squeak VM, Simulated. The Slang code running in the Squeak VM interpreting the image is about 1000 times slower. The system is hardly usable like this, but it is a valuable means to debug the VM with the Smalltalk tools.

SPy VM in C. The result of our written VM after a week of intensive development is not at all bad. It runs at about a tenth the speed of the Squeak VM. This particular version was translated to C, using PyPy's generational GC and profile-guided optimizations.

SPy VM on the CLI. SPY translated to CLI (.NET Common Language Infrastructure) bytecode and running that on Mono is a significant factor slower than translating SPY to C. We assume that this is partially due to PyPy's CLI backend rendering some RPython constructs inefficiently.

SPy VM, Simulated. This is SPY running untranslated on top of CPython[6] (the normal Python interpreter). Similar to running Slang code simulated on another Squeak, this is unusably slow but very useful for testing and debugging.

Potato. The VM running on Java is amazingly fast. Certainly this also has to do with the experience of the author with implementing other Smalltalk VMs.

Pepsi Smalltalk VM. The Squeak VM written with Pepsi is rather slow. The reason for this is that the VM is written in a highly dynamic Smalltalk-like language, which requires a repetition of lookups and message send per single bytecode in the interpreting VM.

Pepsi Compiler. Eliminating these lookups through the compilation of the code down to machine language, brings a huge performance boost. Currently this does not happen automatically through a JIT compiler, but it can be simulated by compiling the Smalltalk code of our benchmark using the Pepsi compiler. This removes the interpretation step from the code, but retains the fully dynamic object model.

OMeta/JS. We ran our OMeta/JS in the Safari Web Browser, as it has one of the fastest JavaScript engines available. We were amazed that it is in the same league as the CLI and Pepsi VM.

VisualWorks. VisualWorks is the fastest Smalltalk VM available today. It uses both sophisticated JIT compiler and memory management technology. The source code is not publicly available.

Hobbes. Running the benchmark reveals that Hobbes is around 100 times slower than the original Squeak VM. However, we have to point out that the Smalltalk-80 user-interface is responsive and comparable in speed to the machines of that time. A reason for that is certainly that Hobbes is running on VisualWorks.

[6] `http://python.org`

5.2 Lines of Code

To give a rough estimate of the comparative complexity of different VM implementations, we included Table 1 with a listing of approximate size of the respective codebase, measured in thousands of lines of code (KLOC).

As shown in Table 1, SPY's RPython source is relatively compact. SPY's RPython source measures only 2600 lines of code, whereas the Slang source for Squeak requires 8900, and even the relatively compact Potato VM weighs in at 4700. This provides further evidence that the higher level of abstraction afforded by RPython is useful for keeping the implementation clean and uncluttered. As discussed in Section 3, we took advantage of a number of RPython features, including annotations, exceptions, and post-processing transformations, to simplify the SPY source and to improve performance. Without such features, SPY would be significantly more complex and more difficult to maintain. Please also note that although our implementation is fairly complete, there are still missing parts (see Section 3.4).

6 Future Work

The section discusses future work regarding SPY. Currently SPY lacks support for several primitives that are needed to make it a realistic replacement of the original Squeak VM. In particular these are primitives for UI and threading. With regard to Squeak's plugin mechanism, we aim to find a way to reuse its interfacing with external functions so we can avoid redoing the work to interface with third-party libraries. When this is done SPY should be a slow but usable replacement for the original Squeak.

Afterwards, we can concentrate on speed optimizations. We plan to implement some straightforward optimizations in the VM, the most obvious example being a method cache. The shadow approach described in Section 3 should make

Table 1. Comparison of VM implementations in KLOC

Implementation	Language	KLOC
Squeak VM	Slang	8.9
Squeak VM (translation)	C	22.8
Spy VM	RPython	2.6
Spy VM (translation)	IL	130.4
Spy VM (translation)	C	187.7
Hobbes VM	Smalltalk	10.0
Potato VM	Java	4.7
Pepsi VM	Pepsi	10.9
Pepsi VM (translation)	C	2.1
OMeta	Javascript	1.4
VisualWorks	C	174.7
#Smalltalk	Smalltalk	7.0
Little Smalltalk	C	4.0
Little Smalltalk	Java	1.8

this straightforward, since the shadows of classes are already kept up-to-date automatically and are thus an obvious place to put a method cache. This should get rid of the most obvious inefficiencies in the current VM.

An area that the PyPy project is currently researching is the automatic generation of just-in-time compilers from interpreters using partial evaluation techniques. The language implementor needs to guide this process with a small number of hints in the interpreter source code. This already works well for PyPy's Python interpreter, where speedups of up to 200 times over normal interpretation can be achieved for simple integer arithmetic [6]. We hope to be able to apply these techniques to SPY as well to get a high-speed VM implementation that could eventually surpass Squeak's performance. This would allow us to get a just-in-time compiler with very little effort, while retaining our easy-to-understand interpreter source code.

While the SPY project specifically tries to use PyPy's toolchain to implement a Squeak VM, it would be worthwhile future project to try to apply some of the ideas of the PyPy project to a pure Squeak setting. This would mean implementing a translation toolchain for a subset of Smalltalk that is higher-level than Slang and then building a VM in it. Doing that would allow it to evaluate PyPy's concepts and to explore the design space for this sort of approach.

7 Conclusion

We have described the implementation details of the SPY project, and provided benchmark results which we believe demonstrate the potential of the SPY project: despite the lack of fundamental optimizations such as a method cache, and the fact that it was coded in a high-level language (complete with garbage collection and other modern amenities), SPY delivers performance competitive with or better than other alternative Squeak implementations.

SPY was developed partly as an experiment to see how suitable the PyPy toolchain would be for a Smalltalk implementation. We found that PyPy is indeed a very useful tool for quickly implementing a virtual machine. The fact that SPY was developed in only one week of development attests to the productivity boost offered by PyPy. By using a high-level language like RPython, and in particular one with support for metaprogramming, we were able to reduce errors and eliminate boilerplate code throughout the system. Furthermore, PyPy's support for translation aspects enabled us to experiment with systematic, low-level optimizations, such as tagged integers, easily and without changes to the interpreter source.

We are confident that with further development, SPY could join Squeak as a realistic platform for Smalltalk development.

Acknowledgments

We thank Orla Greevy for her careful review of a draft of this paper. The members of the Software Composition Group, University of Bern, gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008), and the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance". Adrian Kuhn acknowledges the support of CHOOSE, the Swiss Group for Object-Oriented Systems and Environments.

References

1. Bolz, C.F., Rigo, A.: Memory management and threading models as translation aspects – solutions and challenges. Technical report, PyPy Consortium (2005), http://codespeak.net/pypy/dist/pypy/doc/index-report.html
2. Denker, M.: Entwurf von optimierungen für squeak, Studienarbeit, Universität Karlsruhe (2002)
3. Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison-Wesley, Reading (1983)
4. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, a practical Smalltalk written in itself. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1997), pp. 318–326. ACM Press, New York (1997)
5. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004) (March 2004)
6. Pedroni, S., Rigo, A.: JIT compiler architecture. Technical report, PyPy Consortium (2007), http://codespeak.net/pypy/dist/pypy/doc/index-report.html
7. Rigo, A., Hudson, M., Pedroni, S.: Compiling dynamic language implementations. Technical report, PyPy Consortium (2005), http://codespeak.net/pypy/dist/pypy/doc/index-report.html
8. Rigo, A., Pedroni, S.: PyPy's approach to virtual machine construction. In: Proceedings of the 2006 conference on Dynamic languages symposium, OOPSLA 2006: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 944–953. ACM Press, New York (2006)
9. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer 39(2), 25–31 (2006)

A Source Code of TinyBenchmarks

```
Number>>tinyBenchmarks
    | t1 t2 r n1 n2 |
    n1 := 1.
    [ t1 := Time millisecondsToRun: [ n1 benchmarkPrimes ].
      t1 < 1000 ] whileTrue: [ n1 := n1 * 2 ].

    n2 := 28.
    [ t2 := Time millisecondsToRun: [ r := 28 benchFibonacci ].
      t2 < 1000 ] whileTrue: [ n2 := n2 + 1 ].
```

```
^ ((n1 * 500000 * 1000) // t1) printString , ' bytecodes/sec; ' ,
  ((r * 1000) // t2) printString , ' sends/sec'
 Number>>benchmarkPrimes
    | size flags prime k count |
    size := 8190.
    1 to: self do: [ :iter |
        count := 0.
        flags := (Array new: size) atAllPut: true.
        1 to: size do: [ :i |
            (flags at: i) ifTrue: [
                prime := i + 1.
                k := i + prime.
                [ k <= size ] whileTrue: [
                    flags at: k put: false.
                    k := k + prime ].
                count := count + 1 ] ] ].
    ^ count

Number>>benchFibonacci
    ^ self < 2
        ifTrue: [ 1 ]
        ifFalse: [
            (self - 1) benchmarkFibonacci
                + (self - 2) benchmarkFibonacci + 1 ]
```

B How to Download and Run the Spy Project

Make sure you are running Python version 2.5 or higher, and checkout the project from subversion

```
> svn co http://codespeak.net/svn/pypy/dist pypy-dist
> cd pypy-dist
```

Now, let's generate some Squeak VMs. Switch to the translation goal folder and run the toolchain

```
> cd pypy/translator/goal
> ./translate.py --gc=generation --batch targettinybenchsmalltalk.py
```

To run the generated executable:

```
> ./targettinybenchsmalltalk-c
```

If you browse the target's Python file, you'll find some fixture code together with a function called `entry_point(argv)`. The fixture code is executed before the toolchain takes over. It may use the full power of Python and is not restricted to RPython. Then, the toolchain is started up, taking the `entry_point` function and the fixture's result as an input, to generate the VM. Therefore, all code eventually called by the entry point must conform to RPython.

Are Bytecodes an Atavism?

Theo D'Hondt

Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2,
B1050 Brussels, Belgium
tjdhondt@vub.ac.be

Abstract. The notion of bytecodes can be traced back to the 60's with BCPL O-codes. These were essentially used to pursue platform independence. Later, with Pascal p-codes and Smalltalk bytecodes the objective shifted to the concept of virtual machines as precursors to dedicated hardware implementations, culminating in Lilith and SOAR. More recently, Java adopted a similar approach, but with the advent of efficient JIT-technology, bytecodes resumed their role as intermediary representation of programs written in some higher level language. It is our conjecture that using bytecodes in this capacity is an atavism, a throwback to times where hardware bytecode machines were the ultimate target. We suggest that the question of an optimal intermediary representation must be raised. In this paper we investigate the exact opposite of the bytecode approach: we define an intermediary notation which is as close as possible to the semantics of the programming language under consideration. It is then a question of applying the correct compiler technology to produce an efficient JIT strategy for generating efficient machine code. A more interesting question addressed here is whether a virtual machine can be built using this strategy that matches a bytecode interpreter in perceived performance, while giving the running program much more control over its execution than is the case in the bytecode approach. We investigate a totally non-compromise approach, where a unified memory architecture is used to host all structures relevant during program execution, including program data structures, program representation, interpreter caches and runtime stacks. We existentially prove that it is possible to build a virtual machine along these lines that can match a bytecode implementation in performance while giving much more "self" control to the running program. Two cases are presented here: the Pico language and virtual machine which were co-designed with the unified memory approach in mind, and a Scheme virtual machine intended to match the performance of PLT-Scheme.

Keywords: Virtual machines, bytecodes, interpreters.

1 A Short History of Bytecodes

BCPL [1] is possibly the very first programming language adopting what we now call *bytecodes*. This was in 1966 – BCPL has been largely forgotten, except for its role as a precursor to the C language. Less well known is the fact that a BCPL compiler generates O-code, which is expressed in a lower level instruction set for a fictitious – today we would say *virtual* – computer. O-code programs may then be transformed to machine code for an existing physical computer. This process is intended to facilitate portability of a high-level language implementation across diverse platforms. Presumably this approach was born from the observation that compilers for high-level languages were hard to write, and conversion from one machine code to another proved to be considerably easier. This is not necessarily a truism today.

Arguably one of the most successful applications of the BCPL approach is the Portable Pascal system, also known as *Pascal-P* [2]. Started in 1973, it led to the highly successful UCSD Pascal environment, possibly the very first really high level programming language available on early personal computers. A Pascal-P compiler generated p-code, which was definitely intended to be interpreted: the original software distribution contained a Pascal version of a generic interpreter. This was probably the first public manifestation of a metacircular system in the context of a conventional programming language.

In the case of Pascal-P and its successor *Modula-2* [3], the nature of the intermediary representation (the *bytecodes*) led to the construction of microcode versions of their respective interpreters. Western Digital released the Pascal Microengine [4] and Niklaus Wirth built the Lilith workstation [5], the hardware of which was capable of directly executing Modula-2 m-code. These endeavours were rapidly overtaken by advances in hardware and software. On the one hand custom hardware proved incapable of matching the performance of generic microprocessors such as the Motorola 68000 and the Intel 8086. On the other hand advances in compiler and translator technology considerably reduced the effectiveness of microcode based interpretation.

Possibly the first language to introduce the term bytecode is *Smalltalk* [6]. Although the designers of Smalltalk advocated the use of a virtual machine (i.e. an interpreter), they explicitly suggested using a microcode implementation[1] for performance reasons. Hence the fairly low-level nature of Smalltalk bytecodes[2]. A microcode version of the Smalltalk interpreter was effectively realized in 1987 (see [7]) – but again performance-wise it proved incapable of competing successfully with the available generic microprocessors. Following this insight, the Smalltalk community became instrumental in pioneering fundamental optimisations for virtual machine implementations in software. To name but a few we refer here to inline caching and just-in-time compilation (see for instance [8]), much of which was pioneered in the context of *Self* [14].

Java is today the most widely known language using a virtual machine and its associated bytecode is the most deeply investigated virtual instruction set – although many other modern machine implementations take a similar approach (Python, Ruby and the Microsoft suite, to name but a few). Java bytecodes have also been conceived with a possible microcode implementation in mind – with too many resulting hardware prototypes to name here. Some of these have managed to take hold – albeit in specifically constrained circumstances. It is safe to state that mainstream Java implementations rely on a software approach involving a (software based) virtual machine and a just-in-time

[1] The microcode store of the ALTO workstation was effectively configured to host part of the Smalltalk bytecode interpreter.

[2] By virtue of Smalltalk's very symple syntax bytecodes and source code aren't that far : only lexical addressing of variables stands in the way of full recovery of Smalltalk source code from bytecodes.

compiler to solve performance issues. Some Java implementations have selected to minimize the role played the bytecode stage and compile to native code as a rule (e.g. the Jikes research virtual machine [9]).

It may be concluded from this short – and necessarily incomplete – history, that in the forty or so years since the first appearance of virtual machines, the role of bytecodes has hardly evolved. Moreover, the original reasons for defining bytecodes close to their counterparts in the real world are no longer necessarily valid. Hence the question posed in the title of this paper: are bytecodes an atavism? Or more concretely: if they really are a throwback, what would be a more reasonable alternative to low-level bytecodes as an intermediary representation in a virtual machine approach? This will be addressed in the next section.

2 Virtual Machine Architectures

The term *virtual machine* is typically associated with the interpretation of an instruction set close to *machine code*. In the previous section we described how implementing a virtual machine moved from a software approach to a microcode approach, and then back to a software approach.

In table 1 on the following page we track the different stages that a program goes through when it is executed by its virtual machine. In a typical pre-processing stage, the source code is transformed into a syntax tree, driven by an abstract grammar for the source language. This syntax tree might be completely internalized and serve as a compiler aid only; or it might be rendered explicit to support syntax driven editors or reflective capabilities of the source language. In a second stage the syntax tree is transformed into virtual machine codes (i.e. *bytecodes*). The distance traveled (i.e. the semantic gap) in going from the first to the second is indicated by Δ_1. The third step involves the interpretation (indicated by the symbol) of the bytecodes using a bytecode interpreter which is itself expressed in the hardware's machine codes. Finally, we presume the existence of on-the-fly optimisation that converts bytecode sequences into a machine code sequence prior to execution (popularly known as just-in-time compilation, or JIT). The symbol Δ_2 in table 1 denotes the semantic gap between the bytecodes and machine code.

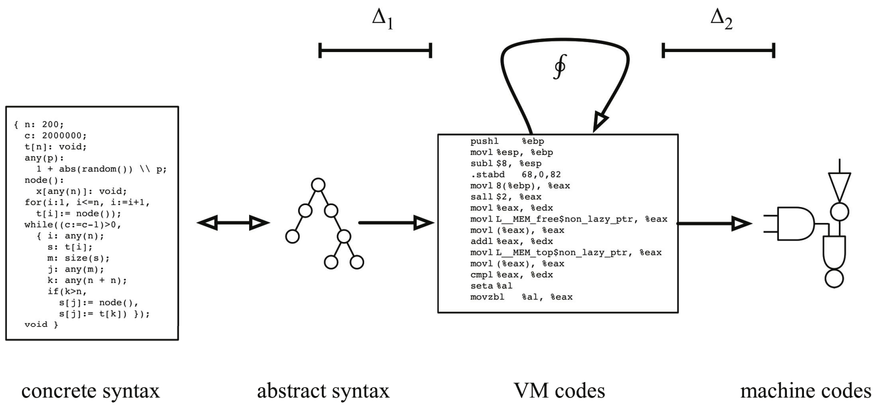
When we try to match current virtual machines with table 1, using the Java VM as the primus, we might venture the conclusion that:

$$\Delta_1 \gg \Delta_2 \approx 0 \tag{1}$$

reflecting the fact that Java bytecodes are very closely related to machine code[3]. This of course raises the question of whether another relationship than the one in (1) could lead to more interesting virtual machine architectures. The common conviction is that performance considerations automatically lead to (1), but no conclusive – other than intuitive – evidence for this conjecture is ever offered.

[3] Some may object that Java (and other) bytecodes incorporate incorporate features such as virtual table support, not available in generic processors. We argue that these still represent (non functional) implementation features.

Table 1. Program execution using a virtual machine



```
{ n: 200;
  c: 2000000;
  t[n]: void;
  any(p):
    1 + abs(random()) \\ p;
  node():
    x[any(n)]: void;
  for(i:1, i<=n, i:=i+1,
    t[i]:= node());
  while((c:=c-1)>0,
    { i: any(n);
      s: t[i];
      m: size(s);
      j: any(m);
      k: any(n + n);
      if(k>n,
        s[j]:= node(),
        s[j]:= t[k]) });
  void }
```

```
pushl    %ebp
movl %esp, %ebp
subl $8, %esp
.stabd   68,0,82
movl 8(%ebp), %eax
sall $2, %eax
movl %eax, %edx
movl L__MEM_free$non_lazy_ptr, %eax
movl (%eax), %eax
addl %eax, %edx
movl L__MEM_top$non_lazy_ptr, %eax
movl (%eax), %eax
cmpl %eax, %edx
seta %al
movzbl   %al, %eax
```

In the next section we will show that an *ex absurdo* proof of this conjecture is not necessarily always true. We shall do so by building an experimental virtual machine that pursues the exact opposite of (1), i.e.

$$\Delta_2 \gg \Delta_1 \approx 0^{4} \tag{2}$$

and show that performance levels are not necessarily impaired by the switch from (1) to (2).

Of course (2) corresponds to the conventional view that an interpreter is the direct instantiation of the target language's semantics. What is different in our approach is the total absence of compromise: we will not increase Δ_1 in the name of performance. Every interpreter implementation – other than purely educational ones – that we are aware of makes allowances for optimisation of internal run-time structures such as environments and execution stacks. We will not subject our virtual machine to this.

The only real constraints driving our design are related to the fact that (any) virtual machine will ultimately have to run on the target machine. This impacts the memory model – typically requiring automatic memory management to be deployed and the runtime model – which requires some kind of support for continuations. Both concerns seem unrelated, but in our approach they will of necessity prove to be mutually dependent.

3 The Pico Virtual Machine

Pico [10] is the name coined for a very compact but expressive language that was co-designed with a virtual machine that implements the precepts proposed in the previous section. In particular, Pico semantics are conceived without any dominant consideration for implementation efficiency. In this section we describe the rationale

[4] This argument is only valid in cases where Δ_1 and Δ_2 represent meaningful values. Consider as an exception the case of *Self*, where both the language and the bytecodes are extremely simple and closely linked.

Table 2. Pico syntactical constructs

	variable	tabulation	application
invocation	`x` variable reference	`t[idx]` table indexing	`f(1, x)` function call
definition	`v: 123` variable definition	`t[10]: x()` table definition	`f(x): x+x` function definition
modification	`v:= 123` variable assignment	`t[10]:= 0` table modification	`f(x):= -x` function redefinition

behind Pico and the architecture of its virtual machine. Subsequently we will discuss its relevance to the discussion at hand.

Pico recognizes 6 disjoint value types: *number*, *fraction*, *text*, *table*, *closure*, *dictionary*[5], *continuation* and *void*. The first three have inline source code representations. All of them can be the target of the Pico evaluator. A total of 9 constructs shape the syntactic structure of Pico. These are enumerated and illustrated in table 2. In addition to these, syntactic sugar is provided to denote compound expressions, compound values, prefix and infix operators and variable length argument lists.

Pico semantics, as defined by a three page metacircular evaluator, is to an important extent inspired by Scheme semantics. However, a number of significant features render Pico different from Scheme:

- Pico does not have special forms – all corresponding constructs are provided as functions;
- Pico does not have a λ-construct – all functions are explicitly named;
- Pico functions have first-class argument lists, stored as tables;
- Pico features *call-by-function*[6] in addition to *call-by-value;*
- Pico has first-class environments;
- All Pico statements, including declarations, are first class Pico expressions;
- Pico does not use lexical addressing and environments are simple association lists;

other than this, Pico has closures, continuations, static scoping, proper tail recursion, i.e. everything that you would expect from a modern dynamic language. Even more so than in other languages such as Scheme, every Pico construct that is reasonable to build is also possible to build.

Below we provide an example to give the reader an idea of Pico-style programming. The function `set` is a higher order function with a variable argument list, which returns a membership test of the numbers provided as arguments. Tables are used to represent ordered nodes in a binary tree: they contain a number and two

[5] A dictionary is a first class lexical environment.

[6] A fairly controversial feature which allows argument expressions to be wrapped as a local function the header of which is specified by the parameter expression.

children. The local functions get and add respectively access and update the tree. The tree is initialized by adding all arguments from the first-class argument table held in Items. At the end a function member is returned; in Scheme an anonymous procedure would be provided, but in Pico we need to name all functions.

```
set @ Items:
  { nbr_idx: 1;
    lft_idx: 2;
    rgt_idx: 3;
    get(Item, Node):
      if(is_table(Node),
        if(Item > Node[nbr_idx],
          get(Item, Node[rgt_idx]),
          get(Item, Node[lft_idx])),
        Item = Node);
    add(Item, Node, Thunk(Tree)):
      if(is_table(Node),
        if(Item > Node[nbr_idx],
          add(Item, Node[rgt_idx], Node[rgt_idx]:= Tree),
          add(Item, Node[lft_idx], Node[lft_idx]:= Tree)),
        if(Item > Node,
          Thunk([Node, Node, Item]),
          if(Item < Node,
            Thunk([Item, Item, Node]))));
    if(size(Items) = 0,
      member(Item): false,
      { TREE: Items[1];
        for(idx: 2, idx<=size(Items), idx:=idx+1,
          add(Items[idx], TREE, TREE:= Tree));
        member(Item): get(Item, TREE) }) }
```

Note the use of the call-by-function Thunk parameter in the add function. This is an example of Pico's call-by-function parameter binding mode; in Scheme, the caller of the add function would have to wrap the argument in a lambda expression.

The Pico virtual machine[7] is built on top of a unified memory model that manages tagged, variable-sized memory chunks. Inaccessible chunks are reclaimed by means of a (non-conservative) compacting mark-and-sweep garbage collector, which comes at the cost of a 1-bit per memory cell overhead. The five page garbage collector is implemented using a five step state machine for the sweep phase, and proves to be extremely efficient[8].

Possibly the most original contribution of our approach is that literally every run-time structure of the Pico evaluator is stored using the unified memory model, with tags to differentiate between the diverse elements. To wit:

- Numbers are represented in an inline format[9];
- Fractions are stored in a raw[10] chunk;

[7] Implemented in C using GCC 4.0.
[8] On a vanilla PC, 100 Mbytes of memory are collected in considerably less than 1 second.
[9] Inspired by the Smalltalk80 virtual machine, using the low order bit in the binary representation.
[10] Raw refers to the fact that the chunk content is exempt from garbage collecting.

- Text is stored in a raw chunk, using a string pool to eliminate duplicates[11];
- Void is represented as an inline binary zero value;
- Tables are stored as variable length memory chunks;
- Closures are stored as parameter list – body – dictionary triplets;
- Dictionaries are stored as string – value association lists;
- Continuations are stored as thread – dictionary pairs;
- Threads are described in the next paragraph;
- The abstract grammar instances corresponding to the nine Pico syntactic variants in table 2 map to correspondingly tagged chunks;

this implies that a running Pico program consists of an evolving graph with variable arity, tagged nodes and possibly cyclical interconnections. Note that garbage collection is equally applicable to conventional program values and to program syntax, environments (dictionaries) and run-time stacks (threads). Safeguards are built in to protect first-class environments and threads from accidental collection when they are embedded in first class closure or continuation values.

Threads are not explicitly reified in this initial version of Pico – their introduction as such is deferred to the eventual introduction of reflective capabilities in Pico. Threads actually serve as link between continuations owned by the evaluator, and they have the following representation:

$$\mathbf{\mathit{thread}} \rightarrow \boxed{\mathit{thread}_{tag}}\boxed{\mathit{thread}_{body}}\boxed{\mathit{thread}_{next}}\boxed{\mathit{thread}_{argument}\quad \cdots} \tag{3}$$

where *thread*$_{body}$ refers to the thread's implementation[12], *thread*$_{next}$ refers to the thread's continuation and the *thread*$_{argument}$'s parametrize the thread. Given that all communication between threads happens via the arguments, the Pico evaluator boils down to the following procedure:

```
evaluate(thread):
  { $thread_{body}$(thread);
    evaluate($thread_{next}$) }
```

The nine semantic routines corresponding to the nine entries in table 2 are set to assemble interconnecting threads that upon execution produce the desired value and side effects in the proper dictionaries.

In practice, a thread is a linked list that is manipulated by the various C-thunks that implement the *thread*$_{body}$'s. These thunks can access the arguments held in the top thread and push onto, pop from, and replace the top thread chunk. A Pico evaluator is a deeply recursive process, part of which uses recursion support in the C-language processor. However, all recursive applications of the evaluator which directly impact the memory model are expressed using the thread network. Garbage collection is preemptively triggered with each thread invocation; the corresponding C-thunk should be viewed as a critical section with respect to memory allocation.

Table 3 gives an idea of the control flow in the continuations that contribute to the Pico evaluator. The nodes represent C-thunks and the arrows represent the control

[11] Pico text is immutable.

[12] At this stage a native C thunk similar to the native C implementation of primitive Pico functions.

Table 3. The Pico evaluation control flow

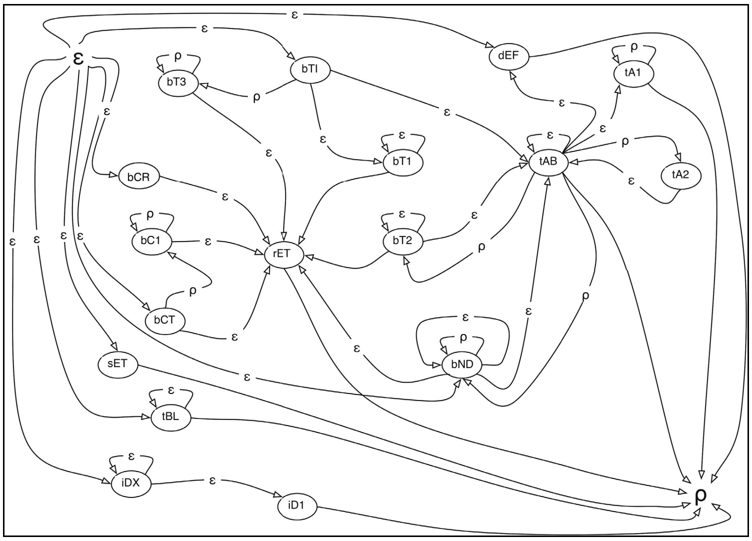


Table 4. Pico virtual machine dimensions

component	header LOC	code LOC	size in Kbytes
Main module	459	341	13
Scanner module	55	377	9
Parser module	15	396	11
Evaluator module	23	1225	13
Printer module	16	783	26
Primitives module	50	2193	52
Dictionary manager	99	40	1
Thread manager	309	384	9
Memory manager	102	203	2

flow between the initial expression ε, and the final result ρ. Edges tagged with an ε-symbol describe the input of a sub-expression from one thunk to another; those tagged with a ρ describe the output of a value. Node labels refer to the names of C functions; for instance `rET` handles the return from a function application and is responsible for storing the caller's environment in the case of a non-tail recursive call. The various nodes `b**` handle all of the parameter binding variants. The `t**` and `i**` nodes handle tables. `sET` and `dEF` speak for themselves. The graph is only included to illustrate that a modest 17 thunks suffice to implement the Pico evaluator.

The dimensions of the complete implementation are given in table 4: slightly over 7000 lines of code against 136 Kbytes of binary code. These values are inflated due to aggressive code unrolling and duplication for performance reasons. Note that the lion's share of the code goes to the implementation of the 50 or so primitive functions defined in the Pico global dictionary, and that the presence of a pretty-printer for debugging purposes explains the printing module's size.

Table 5. A simple Pico benchmark

```
{ QuickSort(V,Low,High):
   { Left: Low;
     Right: High;
     Pivot: V[(Left + Right) // 2];
     Save: 0;
     until(Left > Right,
       { while(V[Left] < Pivot, Left:= Left+1);
         while(V[Right] > Pivot, Right:= Right-1);
         if(Left <= Right,
           { Save:= V[Left];
             V[Left]:= V[Right];
             V[Right]:= Save;
             Left:= Left+1;
             Right:= Right-1 }, void) });
     if(Low < Right, QuickSort(V, Low, Right), void);
     if(High > Left, QuickSort(V, Left, High), void) };
  seed: 0;
  V[20000]: seed:= (seed + 4253171) \\ 1235711;
  QuickSort(V,1,size(V)) }
```

```
{ n: 50000;
  sieve[n]: true;
  multiple: 0;
  index: 0;
  for(index:= 2,
      index<=trunc(sqrt(n)),
      index:=index+1,
      if(sieve[index],
         { multiple:= 2*index;
           while(multiple <= n,
                 { sieve[multiple]:=false;
                   multiple:=multiple+index }) },
         void));
  for(index:= 1, index <= n, index:= index+1,
    if(sieve[index], display(index, eoln), void)) }
```

```
{ fib(n):
    if(n>1, fib(n-1)+fib(n-2), 1);
  fib(25) }
```

Table 6. Pico benchmark results

test case	**PLT Scheme**	**Pico**
Quicksort	1.372s	1.237s
Eratosthenes	0.380s	0.366s
Fibonacci	0.436s	0.648s

All in all we can conclude that a Pico virtual machine implemented according to the guidelines set out in this section is a very modest piece of software, certainly in comparison to what is generally observed; and we call attention to the fact that this implementation covers all components of the Pico read-eval-print cycle, with the full complement of primitives, and all of the advanced features such as first class closures, continuations and tail recursion optimisation.

We decided to test the Pico implementation against the well-known PLT Scheme environment[13] [11], first of all because of its bytecode orientation, and secondly because the proximity in spirit of Pico to Scheme. We assembled a simple benchmark based on three very simple and well-known algorithms: a quicksort of 20000 integers, an Eratosthenes sieve of size 50000 and a non-tail recursive computation of the 25th Fibonacci number. The corresponding Pico code is shown in table 5 and the Scheme code is left to the imagination of the reader. The results[14] are listed in table 6 and they do not disprove the fact that Pico is a match for PLT Scheme performance-wise.

In the next section we will list some of the techniques that led to these results which we remind the reader, were obtained in spite of our no-compromise approach.

4 Some Successful Optimisations

Two non-mutually exclusive approaches should be considered when meeting the demand for optimising a virtual machine implementation. First of all, the (difficult) choice in favour of C as an implementation language leads us to the exploitation to

[13] To level the playing field, we switched off the JIT stage.

[14] As obtained in [12] in 2004; since then successive releases of PLT Scheme have changed the balance somewhat.

their fullest extent of optimisation options offered by the selected C language development environment[15]. Second, all efforts must be undertaken to equip the virtual machine with domain specific algorithmically inspired optimisations. In the first category we mention – without intending to be complete – the following techniques:

- *Selective code duplication*: for instance in the thread manager, the basic stack operations are duplicated for each possible number of arguments in the thread nodes;
- *Implicit code duplication*: critical functions are moved to the header files so as to improve compiler access to these function's implementation;
- *Static function declarations*: static function calls are compiled into simple branch instructions;
- *Static tail recursion*: in the presence of deeply recursive code, a tail recursive style often gives the compiler significant information for optimising the generated code;
- *Invariant code permutation*: knowledge about frequency of execution of particular statements (in for instance then/else clauses of conditionals) can lead to optimisations via code interchanges;

and these can significantly improve the runtime of the virtual machine – or any C-program for that matter.

In the second category we will describe three very effective algorithms. The analysis leading to these algorithms is essentially based on the following observations:

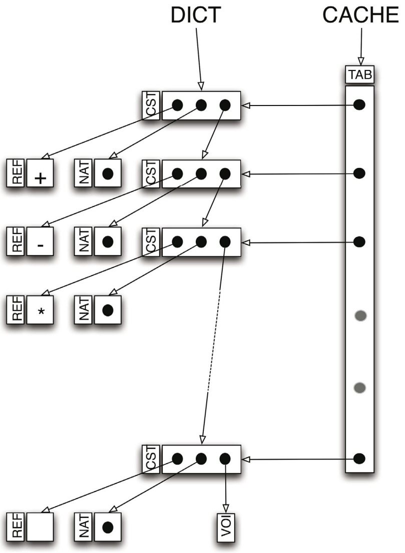
- Although conceptually very clean, the Pico environment structure based on dictionaries, i.e. association lists, is very inefficient. In particular, retrieving a primitive operation requires the sequential traversal of a list before reaching the proper location;
- The straightforward adoption of a unified memory model carries a penalty: very memory intensive actions of the virtual machine, such as thread operations, tend to overload the garbage collector;
- In a similar vein, the allocation of dictionary entries for local variables during function application should not count on garbage collection only in order to deallocate the occupied storage after the function returns;

these three categories of actions – and they are only three of several – impact the performance of the Pico virtual machine to a significant extent. And even if the Pico garbage collector proves to be extremely efficient, this performance drop can only be compensated by a significant increase in storage. In the remainder of this section, we list the solutions that were adopted in the Pico virtual machine to address these three issues.

First optimization: dictionary caching.
In table 7 we describe a dictionary cache as implemented in the Pico virtual machine. The set of primitive variable entries – some fifty of them – form the common trunk of

[15] Which in the case of GCC 4.0 are pretty impressive.

Table 7. A dictionary cache



all dictionaries handled by the Pico evaluator. Since these are created during the initialisation of the virtual machine, we can store them in privileged locations, invariant to garbage collection. If we store them at equidistant locations, we can even use a simple cache vector to guarantee O(1) access to primitive values. Since the top of any dictionary only lists the lexically scoped variables before reaching the common trunk, this proves to be very effective.

Second optimization: thread caching.
In table 8 we describe a thread cache as a kind of simple memory management system: free lists are maintained per allowable thread node size. This variation in size is essentially determined by the number of arguments in (3)[16]. Every thread push operation will retrieve a fresh thread chunk from the appropriate free-list, unless this is empty; in this case memory is explicitly allocated. Every thread pop operation will cause the thread chunk to be added to the appropriate free-list, unless the thread happens to be caught in a first class continuation[17]. This optimisation indeed improves the performance of the virtual machine, but the most dramatic gain is in storage requirements. Introducing a thread cache is the major key to producing a virtual machine that can adopt to very tight memory constraints. On the other hand, the cache is transparent to the rest of the Pico system, implying that the no-compromise principle remains valid.

Third optimization: thread caching.
Table 9 requires some explanation: it describes the creation and reclamation of storage during function application. THREAD refers to the current thread-stack, the

[16] In the Pico virtual machine threads can have up to 7 arguments.
[17] Indicated by a marker bit in the thread chunk header.

Table 8. Thread caching

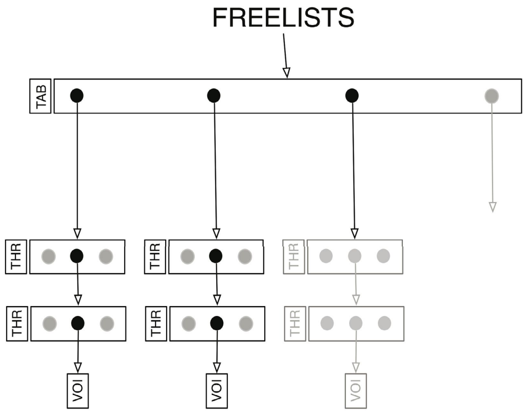
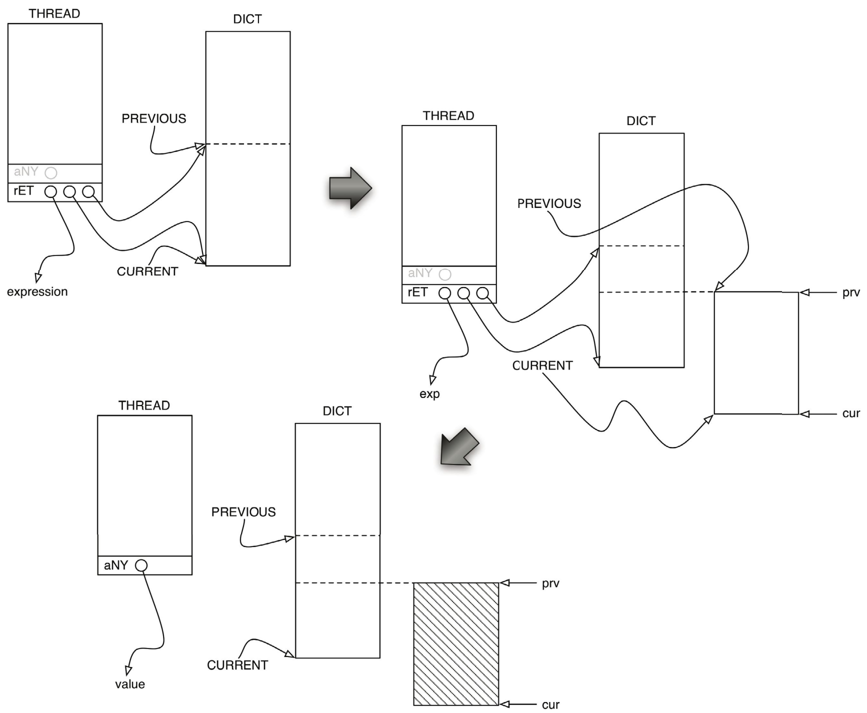


Table 9. Local dictionary handling



top of which contains a return-thread. DICT refers the dictionary (environment) in effect immediately before the function was invoked. The rET thread node is made to hold the CURRENT and PREVIOUS pointers that delimit the current local scope in the current environment. Consequently, these are made to point to the newly allocated environment frame (delimited by cur and prv) in order to evaluate expression in the correct environment. The two environment sections are juxtaposed in order to illustrate that below the bottom hashed line part of the caller’s and the callee’s bindings coexist while above the same line they are shared. Finally, when control returns from the function, CURRENT and PREVIOUS are restored and the hatched area can be reclaimed — provided it isn’t locked by a closure or a continuation. Table 9 only represents one of several possible situations — it is included to provide an idea of the complexity of an environment cache for the evaluator. Note that this process is more complicated than usual because of Pico’s environment architecture based on association lists.

5 The PicoScheme Virtual Machine

In this very short section we want to communicate some initial results from an experiment that is under way. Using the Pico approach as a model, we conceived a Scheme virtual machine called PicoScheme that implements the R5RS [13] standard. At the time of writing, following features remain to be implemented:

- Quasi-quoting;
- Define-syntax and let-syntax;
- The full numerical tower;
- About 40% of less essential primitive functions;
- Full garbage collection support;

but all crucial features, including first class closures and continuations and proper tail recursion, are in place. This is sufficient to make meaningful comparisons with PLT Scheme.

The PicoScheme virtual machine was conceived, designed and implemented with the Pico virtual machine as a model. However, some Scheme-specific mechanisms were necessary. As a language, Scheme is not completely without compromises in the sense mentioned earlier on in this paper. Let's mention a number of them:

- Define is not a first-class Scheme form: it is essentially syntactic sugar to facilitate the optimisation of environment structures and function application;
- Some Scheme statements have a different meaning when used in a top-level position;
- Scheme environments are *not* first class;
- Special forms are definitely intended to be custom compiled;

and these impact the architecture of a virtual machine, unless we explicitly do not want to use their potential for helping optimise the code.

We decided to introduce an extra translation step (a *compiler* in addition to a *reader*) and to opt for lexical addressing. The layout of the PicoScheme virtual machine is depicted in table 10. Scheme programs are first *read* into s-expressions which are then *compiled* to an abstract syntax tree prior to execution. So far nothing new. But the abstract grammar which structures program trees is explicitly used as instruction set for the virtual machine. Again, all program and runtime structures are

Table 10. The PicoScheme virtual machine

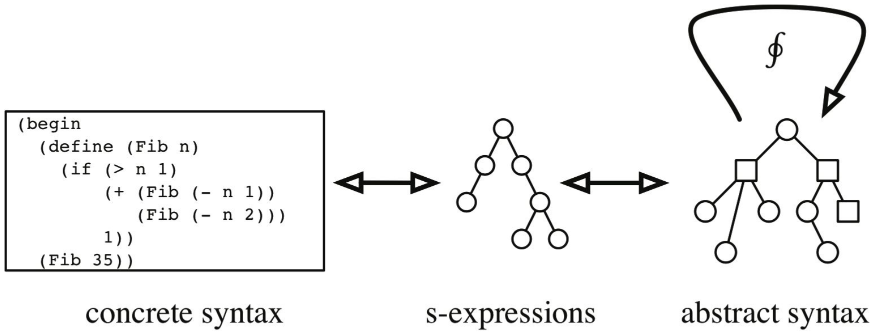


Table 11. PicoScheme benchmark results

test case	**PLT Scheme**	**PicoScheme**
Quicksort	5.210s	5.077s
Eratosthenes	3.589s	2.470s
Fibonacci	6.488s	6.041s

represented in a single unified memory model, totally in line with the Pico virtual machine. And although Scheme variables are converted to lexical addresses, this does not necessarily impair the potential for Scheme programs to reflect over their structure and behaviour[18].

In its current state, the PicoScheme virtual machine counts about 18000 lines of code for 260 Kbytes of binary code; it is expected that completing the implementation will require another 5000 lines of code. This is significantly larger than the Pico virtual machine, caused by the fact that Scheme has considerably more extensive semantics than Pico. This is reflected in the size of the abstract grammar: Pico requires only 17 productions, against 43 for PicoScheme.

We used a similar benchmark to the one in table 6 dating back to 2004; we used higher numbers to have longer-running – and presumably more dependable – results. We performed a quicksort of 500000 integers, an Eratosthenes sieve of size 5000000 and a non-tail recursive computation of the 35th Fibonacci number. Tests were performed on a 2.8GHz Core 2 Duo Intel processor with sufficient memory to avoid triggering the garbage collector. The PicoScheme virtual machine was compiled using GCC 4.0 using -O3 code optimisation. We used version 372 of PLT Scheme and disabled debugging[19], profiling and JIT[20].

Table 11 gives the results, indicating a slightly better performance of PicoScheme over PLT Scheme. Note that during the development of PicoScheme we stopped

[18] Environments are represented as two-level vectors; these vectors are to all intents and purposes equivalent to standard Scheme vectors.

[19] Giving PLT Scheme the benefit of the doubt, since the nature of the PicoScheme virtual machine implies the default presence of debug information; switching on debugging in PLT Scheme results in a performance loss of 130% to 150%.

[20] Activating JIT results in a disappointing 50% to 65% performance gain.

optimising the code when we reached PLT Scheme performances. This does not imply that PicoScheme cannot be even further optimised.

6 Conclusions

In this paper we set out to describe two experiments, which simply stated explore the change of view from situation (1) to situation (2). In the first experiment we co-designed and built from the ground up a language and its virtual machine that maximally respects (2). We showed how smart optimisations – which use the full extent of knowledge about the language's semantics – can compensate for the intrinsic inefficiency of the high level intermediate representation (the bytecodes). In a second experiment we took as a reference PLT Scheme, a very popular Scheme virtual machine that is implemented using the standard bytecode approach; inspired by Pico we built a PicoScheme virtual machine that – the same as PLT Scheme – implements R5RS [13]. PicoScheme combines the principles behind statement (2) with the language level features and optimisations specific to Scheme. The resulting implementation is therefore more extensive than the Pico virtual machine in order to support lexical addressing, quasi-quoting, macro's &c.

The main contribution of this paper – other than producing a virtual machine for Pico and a new virtual machine for Scheme – is an indication that it is possible to build efficient virtual machines with much more expressive intermediary codes than the bytecodes that are generally used. PicoScheme has been optimised to match the performance of PLT Scheme; both implement exactly the same language, therefore a comparison is meaningful. The initial conclusion is simple: in the absence of a valid reason for limiting the level of expressiveness of the virtual machine's instruction set, we can endeavour to vary this expressiveness without impacting the overall performance of the system.

What this paper doesn't do is explore the potential of the principle held in (2). It is not hard to speculate that a custom virtual machine that maximizes the expressiveness of its instruction set for the target language favours reflection – in the meta-programming sense – of a running program on its execution. At the very least it should improve the testing and debugging phase of programming development. These are subjects for further research.

Of course this paper does not do sufficient credit to the effort that has gone into the building of the Pico and PicoScheme virtual machines. It should be viewed as a first step in publishing the results of a major undertaking. Expect seeing specific results in much more detail and of course also expect the various software and other artifacts to become available.

This paper does not propose a universal proof: it doesn't state that a conventional bytecode approach is wrong. It only states that in the absence of justification for using low-level bytecodes it is worthwhile investigating the alternatives. Conversely, if one considers run-time intervention in the execution of a program, it is probably a good idea to avoid intrumenting bytecodes and go for an approach such as advocated here.

Finally: our apologies to all those who didn't find in this paper the things that they expected to see when the topic of virtual machines is covered. We had to leave out a lot of material in order to limit the number of pages. All omissions are intentional and nobody's responsibility but the author's.

References

1. Richards, M.: The Implementation of CPL-like programming languages. Phd thesis. Cambridge University, Cambridge (1966)
2. Barron, D.W. (ed): PASCAL - The Language and its Implementation. ISBN 0-471-27835-1 (1981)
3. Wirth, N.: Programming in Modula-2, 3rd edn. Springer, Heidelberg (1985)
4. Knudsen, S.E.: Medos-2 A Modula-2 Oriented Operating System for the Personal Computer Lilith. Dissertation ETH No 7346 (1983)
5. Pascal MICROENGINE Computer: User manual. The MICROENGINE Company (1979)
6. Design Principles Behind Smalltalk. BYTE Magazine, McGraw-Hill (1981)
7. Bush, W.R., Samples, A.D., Ungar, D., Hilfinger, P.N.: Compiling Smalltalk-80 to a RISC. IEEE Computer Society Press, Los Alamitos (1987)
8. Deutsch, L.P., Schiffman, A.M.: Efficient Implementation of the Smalltalk-80 System. ACM Symposium on Principles of Programming Languages (1984)
9. Jikes Research Virtual Machine home page, `http://jikesrvm.org/`
10. Pico home page, http://pico.vub.ac.be/
11. PLT Scheme home page, `http://www.drscheme.org/`
12. De Meuter, W., D'Hondt, T., Dedecker, J.: Scheme for Mere Mortals. In: Malenfant, J., Østvold, B.M. (eds.) ECOOP 2004. LNCS, vol. 3344. Springer, Heidelberg (2005)
13. Kelsey, R., Clinger, W., Rees, J.: Revised^5 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notices 33(9) (1998)
14. Ungar, D., Smith, R.B.: Self. History of Programming Languages archive. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. ACM, New York (2007)

Author Index