EBERHARD-KARLS-UNIVERSITÄT TÜBINGEN
WILHELM-SCHICKARD-INSTITUT FÜR INFORMATIK
Arbeitsbereich Programmierung — Prof. Dr. H. Klaeren

Übersetzung einer einfachen Programmiersprache

Herbert Klaeren

7. Februar 1995

Inhaltsverzeichnis

Einführung

Wir definieren eine Teilmenge von Pascal[1] durch präzise Angabe der abstrakten Syntax, der kontextsensitiven Nebenbedingungen („statische Semantik") und der denotationellen Semantik. Außerdem wird die Übersetzung dieser Programmiersprache in einen Stapelmaschinen-Zwischencode beschrieben.

Die Beschreibung erfolgt auf allen Ebenen funktional im Sinne von [5] bzw. [10, 9, 4]. Sie ist prinzipiell geeignet für eine Implementierung in SRDL [6]; attraktiver ist jedoch eine gemischte Implementierung im Stile des „multi-paradigm programming" [11, 8].

Als Zielsprache verwenden wir einen stapelorientierten Zwischencode, den D-Code aus [3, 2]. Dieser Zwischencode wird in einer Familie von Modula-2-Compilern kommerziell eingesetzt. Aus didaktischen Gründen weichen wir an einigen Stellen geringfügig vom echten D-Code ab.

Wir bauen die Sprache hier inkrementell nach einem gewissen *Schalenmodell* auf, um jeweils aufzuzeigen, welche Konzepte zur Semantikbeschreibung und Übersetzung erforderlich sind. Wir beginnen also mit einer Sprache, die nur Deklarationen und Ausdrücke über einem einzigen Datentyp (ganze Zahlen) kennt und fügen sukzessive Anweisungen, parameterlose Prozeduren, Prozedurparameter und Datentypen sowie Datenstrukturen hinzu.

Notation

Wir benötigen einige algebraische Grundlagen (s. etwa [6]) und eine gewisse Vertrautheit mit denotationeller Semantik (s. etwa [1]). Es bezeichne $A \to B$ die Menge der partiellen Abbildungen von A nach B. Für $f \in A \to B$ bezeichne $f[a \mapsto b] \in A \to B$ diejenige Abbildung mit

$$f[a \mapsto b](x) \stackrel{\text{def}}{=} \begin{cases} b & \text{falls } x = a \\ f(x) & \text{sonst} \end{cases}$$

Wir benutzen auch die Schreibweise $f[a \mapsto b;\ c \mapsto d]$, wenn keine Namenskonflikte dabei auftreten und auch $f[a_i \mapsto b_i \mid i \in I]$.

Wie in der denotationellen Semantik allgemein üblich, verwenden wir den Schönfinkel-Isomorphismus, um alle Funktionen einstellig zu machen. Die partielle Abbildung

$$f : A \times B \to C$$

definiert man deshalb als

$$f : A \to (B \to C)$$

und schreibt dies in der Form $f : A \to B \to C$, d.h. „$\to$" ist rechtsassoziativ.

Für jedes A^w, $w = w_1 \dots w_n$ verwenden wir die Projektionen $\pi_i^w : A^w \to A^{w_i}$ mit $\pi_i^w(a_1, \dots, a_n) = a_i$, $i \in \{1, \dots, n\}$. Umgekehrt verwenden auch das *Bildprodukt*: Für Abbildungen

$$f_i : A^s \to A^{w_i}, \quad i \in \{1, \dots, n\}$$

definiere

$$\begin{aligned} [f_1; \dots; f_n] : A^s &\to A^w \\ \text{durch } [f_1; \dots; f_n](a) &\stackrel{\text{def}}{=} (f_1(a), \dots, f_n(a)) \end{aligned}$$

Für jeden Konstruktor $c : A^w \to A^s$ betrachten wir die zugehörigen Selektoren $\varsigma_i^c : A^s \to A^{w_i}$ mit $\varsigma_i^c(c(a_1, \dots, a_n)) = a_i$. Meist werden wir allerdings den oberen Index der Selektoren, der den Konstruktor angibt, weglassen; dies vor allem dann, wenn die Sorte s überhaupt nur einen Konstruktor besitzt.

[1] eigentlich von jeder beliebigen Pascal-ähnlichen Sprache

Abstrakter Datentyp Liste

Wir verwenden durchgängig den abstrakten Datentyp List(A), mit den folgenden Operationen:

datatype List(A);

sorts List, A;

constructors
nil: $\rightarrow$ List;
atom: A $\rightarrow$ List;
cons: A $\times$ List $\rightarrow$ List;

operations
hd: List $\rightarrow$ A; *(* Erstes Element *)*
elem: $\mathbb{N}$ $\times$ List $\rightarrow$ A; *(* n-tes Element *)*
tl: List $\rightarrow$ List; *(* Restliste *)*
null: List $\rightarrow$ Bool; *(* Ist Liste leer? *)*
len: List $\rightarrow$ $\mathbb{N}$; *(* Länge der Liste *)*
member: A $\times$ List $\rightarrow$ Bool; *(* Ist a in l ? *)*
append: List $\times$ List $\rightarrow$ List; *(* Anhängen *)*
end

Statt $\mathsf{cons}(a, l)$ schreiben wir grundsätzlich nur $a :: l$, statt $\mathsf{member}(a, l)$ immer $a \in l$. Analog erlauben wir uns $a \notin l$ als Abkürzung von not $\mathsf{member}(a, l)$. Statt $\mathsf{elem}(i, l)$ schreiben wir meist einfach l_i; dementsprechend schreiben wir eine Liste l auch in der Form $[l_1, \ldots, l_n]$. Auch umgekehrt definieren wir eine Liste l gelegentlich einfach dadurch, daß wir die l_i angeben; dann ist die Liste als $l = [l_1, \ldots, l_n]$ gegeben.

Außerdem machen wir Gebrauch von den Funktionalen

$$\begin{array}{rcl} \mathsf{map} : (A \rightarrow B) & \rightarrow & \mathsf{List}(A) \rightarrow \mathsf{List}(B) \\ & \text{mit} & \\ \mathsf{map}(f)(\mathsf{nil}) & \stackrel{\text{def}}{=} & \mathsf{nil} \\ \mathsf{map}(f)(a :: l) & \stackrel{\text{def}}{=} & f(a) :: \mathsf{map}(f)(l) \end{array}$$

sowie

$$\begin{array}{rcl} \mathsf{foldr} : (A \times B \rightarrow B) \times B & \rightarrow & \mathsf{List}(A) \rightarrow B \\ & \text{mit} & \\ \mathsf{foldr}(f, b)(\mathsf{nil}) & \stackrel{\text{def}}{=} & b \\ \mathsf{foldr}(f, b)(a :: l) & \stackrel{\text{def}}{=} & f(a, \mathsf{foldr}(f, b)(l)) \end{array}$$

Wir haben sehr viel mit partiellen Abbildungen zu tun und schreiben dabei $f(x) = \perp$, falls $f(x)$ nicht definiert ist.

Typographische Konventionen

Wir notieren in

Fettschrift Schlüsselwörter der Compiler-Beschreibungssprache

Sans Serif Kontruktoren der abstrakten Syntax, semantische Konstrukte und Übersetzungsfunktionen

`Schreibmaschine` Maschinencode-Instruktionen

1 PLE—Programmiersprache mit Ausdrücken

1.1 Abstrakte Syntax

$\mathbb{Z}$: Die Menge der *ganzen Zahlen*

Ide : Eine (reguläre) Menge von *Bezeichnern*

Decl : Die Menge der *Deklarationen* mit den Konstruktoren:

$$\begin{array}{rcl} \textsf{Decl} & : & \textsf{mk-Decl(List(Const-Decl),List(Var-Decl))} \\ \textsf{Const-Decl} & : & \textsf{mk-Const-Decl(Ide,}\mathbb{Z}\textsf{)} \\ \textsf{Var-Decl} & : & \textsf{mk-Var-Decl(Ide)} \end{array}$$

Op : Eine Menge von *Operationssymbolen*, Op $= \{+, -, *, /\}$

Exp : Die Menge der *Ausdrücke*; diese wird durch die folgenden Konstruktoren beschrieben:

$$\begin{array}{rcl} \textsf{Exp} & : & \textsf{mk-Lit-Exp(}\mathbb{Z}\textsf{)} \\ & & \textsf{mk-Const-Exp(Ide)} \\ & & \textsf{mk-Var-Exp(Ide)} \\ & & \textsf{mk-infix-Exp(Exp,Op,Exp)} \end{array}$$

Prog : Die Menge der *Programme*:

$$\textsf{Prog} \quad : \quad \textsf{mk-Prog(Decl,Exp)}$$

Ein Beispiel soll zeigen, in welcher Weise die abstrakte Syntax von der konkreten Syntax abstrahiert:

Abstrakte Syntax

```
mk-Prog(mk-Decl(mk-Const-Decl(x, 4711) :: nil,
                mk-Var-Decl(y) :: nil),
        mk-infix-Exp(mk-Lit-Exp(23),
                     *,
                     mk-infix-Exp(mk-Const-Exp(x),
                                  +,
                                  mk-Var-Exp(y))))
```

Konkrete Syntax 1

```
program beispiel;
const x=4711;
var y: Integer;
begin
   ReadLn("Bitte y eingeben: ",y);
   WriteLn("Ergebnis ist: ", 23*(x+y))
end.
```

Konkrete Syntax 2

```
#include "stdio.h"
#define x 4711
main ()
```

```
{
   int y;
   printf("Bitte y eingeben: ");
   scanf("%d",y);
   printf("Ergebnis ist: %d\n",23*(x+y) );
}
```

1.2 Semantische Bereiche

$\mathbb{Z}$ die Menge der *ganzen Zahlen*

$\mathsf{Loc} = \mathbb{N}$ eine abzählbar unendliche Menge von *Speicherplätzen (Adressen)*

Store die Menge der *Speicherzustände*, $\mathsf{Store} \stackrel{\text{def}}{=} \mathsf{Loc} \to \mathbb{Z}$

Kind die Menge der *syntaktischen Typen*, $\mathsf{Kind} \stackrel{\text{def}}{=} \{\mathsf{const}, \mathsf{var}\}$.

Env die Menge der *Umgebungen*, $\mathsf{Env} \stackrel{\text{def}}{=} \mathsf{Ide} \to \mathsf{Kind} \times (\mathbb{Z} \cup \mathsf{Loc})$. Es bezeichne $\rho_\emptyset$ die leere Umgebung.

(Da wir $\mathsf{Loc} = \mathbb{N}$ gesetzt haben, erübrigt sich im Grunde die Vereinigung $\mathbb{Z} \cup \mathsf{Loc}$. Wir können diese jedoch durchaus auch als disjunkte Vereinigung verstehen, obwohl auch dies nicht nötig ist, da die Kind-Komponente von Umgebungseinträgen die gewünschte Unterscheidung vornimmt. Die gewählte Schreibweise soll lediglich andeuten, daß hier sowohl Werte als auch Adressen vorkommen können.

1.3 Semantik von Deklarationen

Zur Beschreibung der „statischen Semantik", d.h. also der kontextsensitiven Syntaxbedingungen brauchen wir zunächst die Semantik von Deklarationen:

$\mathcal{D}$ die Semantik von Deklarationen: $\mathcal{D} : \mathsf{Decl} \to \mathsf{Env} \to \mathsf{Env}$ mit

$$
\begin{aligned}
\mathcal{D}[\![\mathsf{mk\text{-}Decl}(d_c, d_v)]\!]\rho &\stackrel{\text{def}}{=} \mathcal{D}_v[\![d_v]\!]\mathcal{D}_c[\![d_c]\!]\rho \\
\mathcal{D}_v[\![\mathsf{nil}]\!]\rho &\stackrel{\text{def}}{=} \mathcal{D}_c[\![\mathsf{nil}]\!]\rho \stackrel{\text{def}}{=} \rho \\
\mathcal{D}_c[\![\mathsf{mk\text{-}Const\text{-}Decl}(i, z) :: w]\!]\rho &\stackrel{\text{def}}{=} \mathcal{D}_c[\![w]\!]\rho[i \mapsto (\mathsf{const}, z)] \quad \text{falls } \rho(i) = \bot \\
\mathcal{D}_v[\![\mathsf{mk\text{-}Var\text{-}Decl}(i) :: w]\!]\rho &\stackrel{\text{def}}{=} \mathcal{D}_v[\![w]\!]\rho[i \mapsto (\mathsf{var}, \mathsf{new}(\rho))] \quad \text{falls } \rho(i) = \bot
\end{aligned}
$$

wobei $\mathsf{new}(\rho)$ das kleinste Element von Loc ist, das bislang noch nicht im Bildbereich von ρ vorkommt. Beachte, daß die Semantik von Konstanten- und Variablendeklarationen voraussetzt, daß kein Bezeichner mehr als einmal deklariert wird.

1.4 Statische Semantik

Für $p = \mathsf{mk\text{-}Prog}(d, e)$, $i \in \mathsf{Ide}$ und $e \in \mathsf{Exp}$ muß gelten:

1. Verschiedenheit der Bezeichner: s.o. 1.3
2. Deklarationszwang und richtiger syntaktischer Typ
   (a) Für jeden Bestandteil $\mathsf{mk\text{-}Const\text{-}Exp}(i)$ von p muß gelten $\mathcal{D}[\![d]\!]\rho_\emptyset(i) = (\mathsf{const}, z)$ für irgendein $z \in \mathbb{Z}$.
   (b) Für jeden Bestandteil $\mathsf{mk\text{-}Var\text{-}Exp}(i)$ von p muß gelten $\mathcal{D}[\![d]\!]\rho_\emptyset(i) = (\mathsf{var}, \alpha)$ für irgendein $\alpha \in \mathsf{Loc}$.

1.5 Semantische Funktionen

Bei der Definition der Semantikfunktionen gehen wir davon aus, daß die Bedingungen der kontextsensitiven Syntax („statische Semantik") erfüllt sind.

$\mathcal{E}$ die Semantik von Ausdrücken: $\mathcal{E} : \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Store} \to \mathbb{Z}$ mit

$$\begin{aligned}
\mathcal{E}[\![\text{mk-Lit-Exp}(z)]\!]\rho\sigma &\stackrel{\text{def}}{=} z\\
\mathcal{E}[\![\text{mk-Const-Exp}(i)]\!]\rho\sigma &\stackrel{\text{def}}{=} \textbf{let } (\mathsf{const}, z) = \rho(i) \textbf{ in } z\\
\mathcal{E}[\![\text{mk-Var-Exp}(i)]\!]\rho\sigma &\stackrel{\text{def}}{=} \textbf{let } (\mathsf{var}, \alpha) = \rho(i) \textbf{ in } \sigma(\alpha)\\
\mathcal{E}[\![(\text{mk-infix-Exp}(l, o, r)]\!]\rho\sigma &\stackrel{\text{def}}{=} f_o(\mathcal{E}[\![l]\!]\rho\sigma, \mathcal{E}[\![r]\!]\rho\sigma)
\end{aligned}$$

wobei f_o die o zugeordnete Operation auf $\mathbb{Z} \times \mathbb{Z}$ ist.

$\mathcal{M}$ die Semantik von Programmen: Jedes Programm enthält nur endlich viele Variablen, denen von $\mathcal{D}[\![\ ]\!]$ die Speicherplätze $\alpha_1, \ldots, \alpha_n$ zugeordnet werden. Definiere dann $\mathcal{M} : \mathsf{Prog} \to \mathbb{Z}^* \to \mathbb{Z}$, mit

$$\mathcal{M}[\![\text{mk-Prog}(d, e)]\!](z_1 z_2 \ldots z_n) \stackrel{\text{def}}{=} \mathcal{E}[\![e]\!]\mathcal{D}[\![d]\!]\rho_\emptyset\sigma_0$$

wobei $\sigma_0(\alpha_i) = z_i$ für $i = 1, \ldots, n$ und $\sigma_0(\alpha) = 0$ sonst.

Zur Initialisierung des Speichers geben wir im Abschnitt über PL0 noch einen Kommentar ab.

1.6 Zwischencode ZE für Ausdrücke

Wir definieren eine abstrakte Stapelmaschine zur Auswertung von Ausdrücken durch:

Zustandsmenge: $SM \stackrel{\text{def}}{=} IP \times S$, wobei

$$\begin{aligned}
IP &\stackrel{\text{def}}{=} \mathbb{N} \quad \textit{(Befehlszeiger, „instruction pointer")}\\
S &\stackrel{\text{def}}{=} \mathbb{Z}^* \quad \textit{(Stapel)}
\end{aligned}$$

Die Stapel schreiben wir so, daß die Stapelspitze rechts ist und die einzelnen Einträge durch Punkte abgetrennt sind. Auf die Elemente von $d \in S$ greifen wir auch in der Form $d(i)$ zu; für $d = d_1.d_2 \ldots d_n$ sei $d(i) \stackrel{\text{def}}{=} d_i$.

Befehlsvorrat: $\Gamma_{\texttt{ZE}} \stackrel{\text{def}}{=} \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{pshLit}\ z, \texttt{pshAdr}\ z, \texttt{deref} \mid z \in \mathbb{Z}\}$

Befehlssemantik: $\mathcal{C} : \Gamma_{\texttt{ZE}} \to SM \to SM$. Jeder Befehl $\gamma \in \Gamma_{\texttt{ZE}}$ bestimmt eine partielle Zustandstransformation $\mathcal{C}[\![\gamma]\!] : SM \to SM$ mit

$$\begin{aligned}
\mathcal{C}[\![\texttt{add}]\!](m, d.z_1.z_2) &\stackrel{\text{def}}{=} (m+1, d.z_1 + z_2)\\
\mathcal{C}[\![\texttt{sub}]\!](m, d.z_1.z_2) &\stackrel{\text{def}}{=} (m+1, d.z_1 - z_2)\\
\mathcal{C}[\![\texttt{mul}]\!](m, d.z_1.z_2) &\stackrel{\text{def}}{=} (m+1, d.z_1 \cdot z_2)\\
\mathcal{C}[\![\texttt{div}]\!](m, d.z_1.z_2) &\stackrel{\text{def}}{=} (m+1, d.\lfloor z_1/z_2 \rfloor) \quad \text{falls } z_2 \neq 0\\
\mathcal{C}[\![\texttt{pshLit}\ z]\!](m, d) &\stackrel{\text{def}}{=} (m+1, d.z)\\
\mathcal{C}[\![\texttt{pshAdr}\ z]\!](m, d) &\stackrel{\text{def}}{=} (m+1, d.z)\\
\mathcal{C}[\![\texttt{deref}]\!](m, d.z) &\stackrel{\text{def}}{=} (m+1, d.d(z)) \quad \text{falls } 0 < z \leq \mathsf{len}(d)
\end{aligned}$$

Die Befehle `pshLit` und `pshAdr` wirken genau gleich; die Unterscheidung wurde nur eingeführt, damit man in einem Zwischencode-Programm erkennbar machen kann, ob eine Zahlenkonstante oder Adreßkonstante gemeint ist.

Programme: Ein ZE-Programm ist eine Liste von ZE-Befehlen; der syntaktische Bereich ZECode sei definiert als

$$\mathsf{ZECode} \stackrel{\text{def}}{=} \mathsf{List}(\Gamma_{\mathtt{ZE}})$$

Einzelschrittsemantik: $\mathcal{S} : \mathsf{ZECode} \to SM \to SM$ sei definiert durch

$$\mathcal{S}[\![p]\!](m, d) \stackrel{\text{def}}{=} \mathcal{C}[\![p_m]\!](m, d)$$

Falls $m < 1$ oder $m > \mathsf{len}(p)$ ist, so ist $\mathcal{S}[\![p]\!](m, d)$ undefiniert. Dies ist natürlich auch dann der Fall, wenn $\mathcal{C}[\![p_m]\!](m, d)$ undefiniert ist.

Iterationssemantik: $\mathcal{M} : \mathsf{ZECode} \to SM \to SM$ sei definiert durch

$$\mathcal{M}[\![p]\!](m, d) \stackrel{\text{def}}{=} \begin{cases} (m, d) & \text{falls} \quad \mathcal{S}[\![p]\!](m, d) \text{ undefiniert} \\ \mathcal{M}[\![p]\!]\mathcal{S}[\![p]\!](m, d) & \text{sonst} \end{cases}$$

Anschaulich gesprochen, hält die abstrakte ZE-Zwischencodemaschine an, sobald einmal die Einzelschrittsemantik undefiniert ist. Dies kann, wie zuvor bereits angedeutet, aus drei Gründen geschehen:

1. Die letzte Instruktion des ZE-Programms wurde ausgeführt.
2. Eine arithmetische Instruktion sollte ausgeführt werden, wobei aber nicht genügend viele Operanden auf dem Stapel lagen, oder es sollte eine nicht existente Adresse dereferenziert werden.
3. Es wurde eine Division durch Null versucht.

Die erste Möglichkeit würden wir als ein reguläres Ende des Programms bezeichnen, während die anderen beiden *Laufzeitfehler* sind. Möglichkeit zwei ist auszuschließen, wenn wir aus syntaktisch korrekten PLE-Programmen Code erzeugt haben, mit der dritten Möglichkeit müssen wir jedoch rechnen.

1.7 Übersetzung von PLE nach ZE

1.7.1 Symboltabelle

Zur Kontrolle der Einhaltung der statischen Semantik und zur Codeerzeugung für Konstanten und Bezeichner verwendet man in Compilern statt der *Umgebung* eine Symboltabelle, die wir im folgenden definieren. Bei dieser einfachen Sprache verhält sich die Symboltabelle genau gleich zu der Umgebung, später wird sich dies ändern.

$$\mathsf{Tab} \stackrel{\text{def}}{=} \mathsf{Ide} \to (\{\mathsf{const}\} \times \mathbb{Z}) \cup (\{\mathsf{var}\} \times \mathbb{N})\,.$$

Es bezeichne $st_\emptyset \in \mathsf{Tab}$ die leere Symboltabelle. Wir möchten jetzt eine semantische Funktion update zur Erstellung einer Symboltabelle definieren, wobei gleichzeitig die Zuordnung von Adressen zu Variablenbezeichnern vorgenommen werden soll. Hierzu definieren wir analog zur Funktion new aus der Semantik eine Hilfsfunktion

$$\mathsf{newloc} : \mathsf{Tab} \to \mathsf{Loc}$$

die jeweils den niedrigsten Speicherplatz zurückgibt, der in der Symboltabelle noch nicht einer Variablen zugeordnet wurde. Dann ist

$$\mathsf{update} : \mathsf{Decl} \to \mathsf{Tab} \to \mathsf{Tab}$$

definiert durch

$$\begin{aligned}
\mathsf{update}(\mathsf{mk\text{-}Decl}(d_c, d_v), st) &\stackrel{\text{def}}{=} \mathsf{v\text{-}update}(d_v, \mathsf{c\text{-}update}(d_c, st)) \\
\mathsf{c\text{-}update}(\mathsf{nil}, st) &\stackrel{\text{def}}{=} \mathsf{v\text{-}update}(\mathsf{nil}, st) \stackrel{\text{def}}{=} st \\
\mathsf{c\text{-}update}(\mathsf{mk\text{-}Const\text{-}Decl}(i, z) :: w, st) &\stackrel{\text{def}}{=} \mathsf{c\text{-}update}(w, st[i \mapsto (\mathsf{const}, z)]) \\
\mathsf{v\text{-}update}(\mathsf{mk\text{-}Var\text{-}Decl}(i) :: w, st) &\stackrel{\text{def}}{=} \mathsf{v\text{-}update}(w, st[i \mapsto (\mathsf{var}, \mathsf{newloc}(st))])
\end{aligned}$$

In echten Compilern wird man die Symboltabelle st und die Angabe des nächsten freien Speicherplatzes nicht als Argument (und Rückgabewert) der Compilerfunktionen führen, sondern als *globale Variablen* des Compilers. update wird dann zu einer Prozedur:

$$\mathsf{update} : \mathsf{Decl} \to \emptyset$$

1.7.2 Übersetzung von Ausdrücken

ZECode-Programme sind formal Listen von ZECode-Instruktionen. Bei der Notation dieser Listen erlauben wir uns einige Freiheit, indem wir Listenelemente und Teillisten einfach durch Zeilensprünge oder notfalls durch Semikolon abtrennen.

Definiere

$$\mathsf{optrans} : O \to \Gamma_{\mathtt{ZE}}$$

durch

$$\begin{aligned}
\mathsf{optrans}(+) &\stackrel{\text{def}}{=} \mathtt{add} \\
\mathsf{optrans}(-) &\stackrel{\text{def}}{=} \mathtt{sub} \\
\mathsf{optrans}(*) &\stackrel{\text{def}}{=} \mathtt{mul} \\
\mathsf{optrans}(/) &\stackrel{\text{def}}{=} \mathtt{div}
\end{aligned}$$

und

$$\mathsf{exptrans} : \mathsf{Exp} \times \mathsf{Tab} \to \mathsf{ZECode}$$

durch

$$\begin{aligned}
\mathsf{exptrans}(\mathsf{mk\text{-}Lit\text{-}Exp}(z), st) &\stackrel{\text{def}}{=} \mathtt{pshLit}\; z \\
\mathsf{exptrans}(\mathsf{mk\text{-}Const\text{-}Exp}(i), st) &\stackrel{\text{def}}{=} \mathbf{let}\ (\mathsf{const}, z) = st(i)\ \mathbf{in} \\
&\phantom{\stackrel{\text{def}}{=}}\ \mathtt{pshLit}\; z \\
\mathsf{exptrans}(\mathsf{mk\text{-}Var\text{-}Exp}(i), st) &\stackrel{\text{def}}{=} \mathbf{let}\ (\mathsf{var}, n) = st(i)\ \mathbf{in} \\
&\phantom{\stackrel{\text{def}}{=}}\ \mathtt{pshAdr}\; n; \mathtt{deref} \\
\mathsf{exptrans}(\mathsf{mk\text{-}infix\text{-}Exp}(l, o, r), st) &\stackrel{\text{def}}{=} \mathsf{exptrans}(l, st) \\
&\phantom{\stackrel{\text{def}}{=}}\ \mathsf{exptrans}(r, st) \\
&\phantom{\stackrel{\text{def}}{=}}\ \mathsf{optrans}(o)
\end{aligned}$$

1.7.3 Übersetzung von Programmen

$$\mathsf{trans} : \mathsf{Prog} \to \mathsf{ZECode}$$

$$\mathsf{trans}(\mathsf{mk\text{-}Prog}(d, e)) \stackrel{\text{def}}{=} \mathsf{exptrans}(e, \mathsf{update}(d, st_\emptyset))$$

Beispiel Für das am Beginn dieses Abschnitts vorgestellte Beispielprogramm ergibt sich zunächst als Ergebnis von update folgende Symboltabelle (hier tatsächlich als Tabelle dargestellt):

x	const	4711
y	var	0

und dann als Ergebnis von exptrans das folgende Programm:

```
pshLit 23
pshLit 4711
pshAdr 0
deref
add
mul
```

1.8 Korrektheit der Übersetzung

Die Korrektheitsbedingung wollen wir nur formulieren, nicht beweisen. Wegen

$$\mathcal{M}[\![P]\!] : \mathbb{Z}^* \to \mathbb{Z} \quad \text{für P} \in \mathsf{Prog}$$

und

$$\mathcal{M}[\![P]\!] : SM \to SM \quad \text{für } P \in \mathsf{ZECode}$$

kann etwa $\mathcal{M}[\![P]\!] = \mathcal{M}[\![\mathsf{trans}(P)]\!]$ nicht erwartet werden. Außerdem müssen wir beachten, daß das Maschinenprogramm durch vorzeitiges Anhalten bei Laufzeitfehler ein (falsches) Resultat in Fällen liefern kann, wo die formale Semantik des PLE-Programms undefiniert ist. Doch auch der umgekehrte Fall ist möglich: Enthält ein Programm n Variablen und wurden nur $m < n$ Argumente übergeben, so enthält eine nicht initialisierte Variable in der formalen Semantik einfach den Wert 0, während das Maschinenprogramm bei der Auswertung von `deref` mit einer zu großen Adresse hängenbleibt.

Wir definieren deshalb Eingabe- und Ausgabeabbildungen, welche die Übersetzung zwischen den beiden semantischen Welten vornehmen, indem wir durch geeignete Indizes die unerwünschten Sonderfälle ausschließen. Im einzelnen ist:

$$\mathsf{in}_i : \mathbb{Z}^* \to \mathrm{SM}$$

$$\mathsf{in}_i(z_1 z_2 \ldots z_n) \stackrel{\text{def}}{=} (1, z_1 z_2 \ldots z_m)$$

mit $m = \max(n, i)$ und $z_j = 0$ für $j > n$ und

$$\mathsf{out}_i : SM \to \mathbb{Z}$$

$$\mathsf{out}_i(m, d.z) \stackrel{\text{def}}{=} z \quad \text{falls } i = m\ .$$

Die Eingabeabbildung bringt die Eingabeargumente und ggf. zusätzliche Null-Werte auf den Stapel der SM, während die Ausgabeabbildung das oberste Element des Stapels ausgibt, falls dieser nicht leer ist. Dabei ist out nur definiert, wenn das Ende des Programms auf normale Art und Weise erreicht wurde.

Satz 1.1 *Für alle $p \in$ Prog, die genau k Variablen enthalten, gilt*

$$\mathcal{M}[\![p]\!] = \mathsf{out}_{\mathsf{len}(\mathsf{trans}(p))+1} \circ \mathcal{M}[\![\mathsf{trans}(p)]\!] \circ \mathsf{in}_k\ .$$

1.9 Alternative Codeerzeugung mit „constant folding“

Üblicherweise erzeugt man keinen Code für die Auswertung von Ausdrücken, die sich bereits zur Übersetzungszeit auswerten lassen. Definiere dazu

$$\textsf{is-const-Expr} : \textsf{Exp} \to \{\mathbf{true}, \mathbf{false}\}$$

durch

$$\begin{aligned}
\textsf{is-const-Expr}(\textsf{mk-Lit-Exp}(z)) &\stackrel{\text{def}}{=} \mathbf{true}\\
\textsf{is-const-Expr}(\textsf{mk-Const-Exp}(i)) &\stackrel{\text{def}}{=} \mathbf{true}\\
\textsf{is-const-Expr}(\textsf{mk-Var-Exp}(i)) &\stackrel{\text{def}}{=} \mathbf{false}\\
\textsf{is-const-Expr}(\textsf{mk-infix-Exp}(l, o, r)) &\stackrel{\text{def}}{=} \textsf{is-const-Expr}(l) \wedge \textsf{is-const-Expr}(r)
\end{aligned}$$

und ändere exptrans wie folgt:

$$\textsf{exptrans}(\textsf{mk-infix-Exp}(l, o, r), st) \stackrel{\text{def}}{=} \begin{cases} \texttt{pshLit}\ z & \text{falls } \textsf{is-const-Expr}(\textsf{mk-infix-Exp}(l, o, r)) \\ & \text{und } f_o(\mathcal{E}[\![l]\!]\rho_1\sigma, \mathcal{E}[\![r]\!]\rho_1\sigma) = z \\ \quad \textsf{exptrans}(l, st) & \text{sonst} \\ \quad \textsf{exptrans}(r, st) & \\ \quad \textsf{optrans}(o) & \end{cases}$$

Dabei sei ρ_1 die relevante Umgebung aus der Semantikdefinition[2] und σ beliebig; die Semantik konstanter Ausdrücke hängt vom Speicher nicht ab.

[2] Eine Abbildung $\textsf{tab2env} : \textsf{Tab} \to \textsf{Env}$ läßt sich auf triviale Weise definieren; dann ist $\rho_1 = \textsf{tab2env}(st)$

2 PLA—Programmiersprache mit Anweisungen

In der ersten Erweiterungsstufe nehmen wir Wertzuweisungen und einige wenige Kontrollstrukturen hinzu. Hier wie auch in den nachfolgenden Abschnitten wollen wir jeweils nur noch auflisten, was in den einzelnen Teilen der Sprach- oder Übersetzerbeschreibung hinzukommt. Auf diese Weise wird die Übersicht erleichtert, was jeweils an „Kosten“ für neue Konzepte entsteht, auch wenn die Übersicht über die ganze Sprachbeschreibung verschleiert wird. Ein Anhang (A) faßt die komplette Beschreibung der Programmiersprache in ihrer komplexesten Version zusammen.

2.1 Abstrakte Syntax

BOp : Eine Menge von *Boole'schen (Infix-) Operationssymbolen*, $\mathsf{BOp} \stackrel{\text{def}}{=} \{\mathsf{and}, \mathsf{or}\}$

Rel : Eine Menge von *Relationssymbolen*, $\mathsf{Rel} = \{<, \leq, >, \geq, =, \neq\}$

BExp : Die Menge der *Boole'schen Ausdrücke*:

$$\begin{array}{lll} \mathsf{BExp} & : & \mathsf{mk\text{-}rel\text{-}BExp(Exp,Rel,Exp)} \\ & & \mathsf{mk\text{-}not\text{-}BExp(BExp)} \\ & & \mathsf{mk\text{-}infix\text{-}BExp(BExp,BOp,BExp)} \end{array}$$

Stmt : Die Menge der *Anweisungen*:

$$\begin{array}{lll} \mathsf{Stmt} & : & \mathsf{mk\text{-}assign\text{-}Stmt(Ide,Exp)} \\ & & \mathsf{mk\text{-}sequence\text{-}Stmt(List(Stmt))} \\ & & \mathsf{mk\text{-}if\text{-}Stmt(BExp,Stmt,Stmt)} \\ & & \mathsf{mk\text{-}while\text{-}Stmt(BExp,Stmt)} \end{array}$$

Prog : Die Menge der *Programme*:

$$\mathsf{Prog} \quad : \quad \mathsf{mk\text{-}Prog(Decl,Stmt,Exp)}$$

2.2 Semantische Bereiche

$\mathsf{Stat} \stackrel{\text{def}}{=} \mathsf{Store} \to \mathsf{Store}$ die *Speichertransformationen*

2.3 Semantik von Deklarationen

— keine Änderung —

2.4 Statische Semantik

2. Deklarationszwang und richtiger syntaktischer Typ
   Für $p = \mathsf{mk\text{-}Prog}(d, s, e)$ und $i \in \mathsf{Ide}$ muß gelten:

   (c) Für jeden Bestandteil $\mathsf{mk\text{-}assign\text{-}Stmt}(i, e')$ von p muß gelten $\mathcal{D}[\![d]\!]\rho_\emptyset(i) = (\mathsf{var}, \alpha)$ für irgendein α.

2.5 Semantische Funktionen

$\mathcal{B}$ die Semantik von Boole'schen Ausdrücken: $\mathcal{B}$: $\mathsf{BExp} \to \mathsf{Env} \to \mathsf{Store} \to \{\mathbf{true}, \mathbf{false}\}$ mit

$$\begin{aligned}
\mathcal{B}[\![\textsf{mk-rel-BExp}(e_1, r, e_2)]\!]\rho\sigma &\stackrel{\text{def}}{=} f_r(\mathcal{E}[\![e_1]\!]\rho\sigma, \mathcal{E}[\![e_2]\!]\rho\sigma) \\
\mathcal{B}[\![\textsf{mk-not-BExp}(b)]\!]\rho\sigma &\stackrel{\text{def}}{=} \neg\mathcal{B}[\![b]\!]\rho\sigma \\
\mathcal{B}[\![\textsf{mk-infix-BExp}(b_1, o, b_2)]\!]\rho\sigma &\stackrel{\text{def}}{=} f_o(\mathcal{B}[\![b_1]\!]\rho\sigma, \mathcal{B}[\![b_2]\!]\rho\sigma)
\end{aligned}$$

Dabei seien f_r, f_o die Interpretationen der Relationen r bzw. Operationen o. Bemerkt sei, daß damit nur scheinbar die Semantik Boole'scher Ausdrücke präzise definiert wurde; die Anwesenheit undefinierter Berechnungen läßt verschiedene Optionen offen, die allerdings erst später im Zusammenhang mit nicht-endenden Berechnungen voll zum Tragen kommen:

1. strikte Boole'sche Operationen:

$$\begin{aligned}
b \wedge \bot &= \bot = \bot \wedge b \\
b \vee \bot &= \bot = \bot \vee b
\end{aligned}$$

2. sequentielle Boole'sche Operationen: *(„short circuit evaluation")*

$$\begin{aligned}
\mathbf{false} \wedge \bot &= \mathbf{false} \\
\bot \wedge b &= \bot \\
\mathbf{true} \vee \bot &= \mathbf{true} \\
\bot \vee b &= \bot \\
\mathbf{true} \wedge \bot &= \bot = \mathbf{false} \vee \bot
\end{aligned}$$

3. parallele Boole'sche Operationen:

$$\begin{aligned}
\mathbf{false} \wedge \bot &= \mathbf{false} = \bot \wedge \mathbf{false} \\
\mathbf{true} \vee \bot &= \mathbf{true} = \bot \vee \mathbf{true}
\end{aligned}$$

sonst undefiniert.

$\mathcal{S}$ die Semantik von Anweisungen: $\mathcal{S}$: $\mathsf{Stmt} \to \mathsf{Env} \to \mathsf{Stat}$ mit

$$\begin{aligned}
\mathcal{S}[\![\textsf{mk-assign-Stmt}(i, e)]\!]\rho\sigma &\stackrel{\text{def}}{=} \sigma[\alpha \mapsto \mathcal{E}[\![e]\!]\rho\sigma] \textbf{ where } (\textsf{var}, \alpha) = \rho(i) \\
\mathcal{S}[\![\textsf{mk-sequence-Stmt}(\textsf{nil})]\!]\rho\sigma &\stackrel{\text{def}}{=} \sigma \\
\mathcal{S}[\![\textsf{mk-sequence-Stmt}(s :: w)]\!]\rho\sigma &\stackrel{\text{def}}{=} \mathcal{S}[\![\textsf{mk-sequence-Stmt}(w)]\!]\rho(\mathcal{S}[\![s]\!]\rho\sigma) \\
\mathcal{S}[\![\textsf{mk-if-Stmt}(b, s_1, s_2)]\!]\rho\sigma &\stackrel{\text{def}}{=} \begin{cases} \mathcal{S}[\![s_1]\!]\rho\sigma & \text{falls } \mathcal{B}[\![b]\!]\rho\sigma = \mathbf{true} \\ \mathcal{S}[\![s_2]\!]\rho\sigma & \text{falls } \mathcal{B}[\![b]\!]\rho\sigma = \mathbf{false} \end{cases} \\
\mathcal{S}[\![\textsf{mk-while-Stmt}(b, s)]\!]\rho\sigma &\stackrel{\text{def}}{=} \begin{cases} \mathcal{S}[\![\textsf{mk-while-Stmt}(b, s)]\!]\rho(\mathcal{S}[\![s]\!]\rho\sigma) & \text{falls } \mathcal{B}[\![b]\!]\rho\sigma = \mathbf{true} \\ \sigma & \text{falls } \mathcal{B}[\![b]\!]\rho\sigma = \mathbf{false} \end{cases}
\end{aligned}$$

$\mathcal{M}$ die Semantik von Programmen: $\mathcal{M}$: $\mathsf{Prog} \to \mathbb{Z}^* \to \mathbb{Z}$, mit

$$\mathcal{M}[\![\textsf{mk-Prog}(d, s, e)]\!](z_1, \ldots, z_n) \stackrel{\text{def}}{=} \mathcal{E}[\![e]\!]\rho_1\mathcal{S}[\![s]\!]\rho_1\sigma_0$$

wobei $\rho_1 = \mathcal{D}[\![d]\!]\rho_\emptyset$ und $\sigma_0(\alpha_i) = z_i$ für $i = 1, \ldots, n$, $\sigma_0(\alpha) = 0$ sonst.

2.6 Zwischencode ZA für Anweisungen

Wegen der **if**- und **while**-Anweisungen brauchen wir jetzt bedingte und unbedingte Sprungbefehle und daher auch die Möglichkeit der Angabe von Sprungzielen. Wir sehen daher die Möglichkeit vor, daß Befehle mit *Marken („labels“)* versehen werden können und nehmen an, daß die abstrakte ZA-Maschine mit solchen Marken umgehen kann. Marken sind für uns dasselbe wie Bezeichner und ein Befehl kann keine, eine oder mehrere Marken haben. Zu jeder Marke soll es höchstens einen Befehl geben, der mit dieser Marke versehen ist.

Wir definieren also eine abstrakte Stapelmaschine zur Auswertung von Anweisungen durch:

Befehlsvorrat:

$$\begin{aligned}\Gamma_{\mathtt{ZA}} &\stackrel{\text{def}}{=} \Gamma_{\mathtt{ZE}} \cup \{\mathtt{assign}, \mathtt{branch}\, l, \mathtt{brFalse}\, l, \mathtt{brTrue}\, l \mid l \in \mathsf{Ide}\}\\ &\quad \cup\{\mathtt{and}, \mathtt{or}, \mathtt{not}, \mathtt{relEQ}, \mathtt{relGE}, \mathtt{relGT}, \mathtt{relLE}, \mathtt{relLS}, \mathtt{relNE}\}\end{aligned}$$

Befehlssemantik: Jeder Befehl $\gamma \in \Gamma_{\mathtt{ZA}}$ bestimmt eine partielle Zustandstransformation $\mathcal{C}[\![\gamma]\!] \in SM \to SM$, wobei die ZE-Befehle wie bei PLE definiert seien und

$$\begin{aligned}
\mathcal{C}[\![\mathtt{assign}]\!](m, d.x.n) &\stackrel{\text{def}}{=} (m+1, d[n \mapsto x]) \quad \text{falls } n \leq \mathsf{len}(d)\\
\mathcal{C}[\![\mathtt{branch}\, l]\!](m, d) &\stackrel{\text{def}}{=} (i_l, d)\\
&\quad\ \text{wobei } i_l \text{ die Nummer der Instruktion mit Marke } l \text{ ist}\\
\mathcal{C}[\![\mathtt{brFalse}\, l]\!](m, d.x) &\stackrel{\text{def}}{=} \begin{cases}(i_l, d) & \text{falls } x = 0\\ (m+1, d) & \text{falls } x \neq 0\end{cases}\\
\mathcal{C}[\![\mathtt{brTrue}\, l]\!](m, d.x) &\stackrel{\text{def}}{=} \begin{cases}(i_l, d) & \text{falls } x = 1\\ (m+1, d) & \text{falls } x \neq 1\end{cases}\\
\mathcal{C}[\![\mathtt{and}]\!](m, d.d_1.d_2) &\stackrel{\text{def}}{=} (m+1, d.d_1 \wedge d_2)\\
\mathcal{C}[\![\mathtt{or}]\!](m, d.d_1.d_2) &\stackrel{\text{def}}{=} (m+1, d.d_1 \vee d_2)\\
\mathcal{C}[\![\mathtt{not}]\!](m, d.d_1) &\stackrel{\text{def}}{=} (m+1, d.\neg d_1)
\end{aligned}$$

Dabei seien die logischen Operationen auf der Grundlage $0 \equiv$ **false**, $1 \equiv$ **true** definiert. Ihren Effekt für andere Argumente als 0 und 1 lassen wir undefiniert.

$$\begin{aligned}
\mathcal{C}[\![\mathtt{relEQ}]\!](m, d.d_1.d_2) &\stackrel{\text{def}}{=} \begin{cases}(m+1, d.0) & \text{falls } d_1 \neq d_2\\ (m+1, d.1) & \text{falls } d_1 = d_2\end{cases}\\
\mathcal{C}[\![\mathtt{relGE}]\!](m, d.d_1.d_2) &\stackrel{\text{def}}{=} \begin{cases}(m+1, d.0) & \text{falls } d_1 < d_2\\ (m+1, d.1) & \text{falls } d_1 \geq d_2\end{cases}
\end{aligned}$$

`relGT`, `relLE`, `relLT` und `relNE` werden analog definiert.

Programme: Der syntaktische Bereich ZACode sei definiert als

$$\mathsf{ZACode} \stackrel{\text{def}}{=} \mathsf{List}(\mathsf{List}(\mathsf{Ide}) \times \Gamma_{\mathtt{ZA}})$$

Einzelschritt- und Programmsemantik werden nicht geändert.

2.7 Übersetzung von PLA nach ZA

2.7.1 Symboltabelle

— keine Änderung —

2.7.2 Übersetzung von Ausdrücken

Definiere

$$\mathsf{reltrans} : R \to \Gamma_{\mathtt{ZA}}$$

$$\mathsf{boptrans} : \{\mathsf{and}, \mathsf{or}\} \to \Gamma_{\mathtt{ZA}}$$

in naheliegender Weise und

$$\mathsf{bexptrans} : \mathsf{BExp} \to \mathsf{Tab} \to \mathsf{ZACode}$$

in ebenso naheliegender Weise analog zu exptrans.

2.7.3 Übersetzung von Anweisungen

Bei der Übersetzung von Anweisungen spezifizieren wir Befehlsmarken, sofern sie benötigt werden, in der in Assemblersprachen üblichen Syntax (`label:`). Wir erlauben uns auch, Marken ohne nachfolgenden Befehl zu erzeugen; diese Marken werden, falls später danach ein Befehl folgt, als zu der Markenliste dieses Befehls gehörig betrachtet. Bei syntaktisch korrekten Programmen, die wir hier wie überall voraussetzen wollen, kann es nicht vorkommen, daß kein Befehl nachfolgt.

Zur Erleichterung der Erzeugung verschiedener Marken setzen wir eine Hilfsfunktion

$$\mathsf{newlabel} :\to \mathsf{Ide}$$

voraus, die bei jedem Aufruf eine neue Marke zurückgibt. (In (pseudo-) funktionalen Sprachen wie LISP, HOPE, ML etc. heißen solche Funktionen „Pseudo-Konstanten".)

Definiere

$$\mathsf{stmtrans} : \mathsf{Stmt} \to \mathsf{Tab} \to \mathsf{ZACode}$$

durch

$$\begin{array}{rcl}
\mathsf{stmtrans}(\mathsf{mk\text{-}assign\text{-}Stmt}(i, e), st) & = & \mathbf{let}\ \ (\mathsf{var}, n) = st(i)\ \mathbf{in} \\
& & \quad \mathsf{exptrans}(e, st) \\
& & \quad \mathtt{pshAdr}\ n \\
& & \quad \mathtt{assign} \\
\mathsf{stmtrans}(\mathsf{mk\text{-}sequence\text{-}Stmt}(\mathsf{nil}), st) & \stackrel{\mathrm{def}}{=} & \mathsf{nil} \\
\mathsf{stmtrans}(\mathsf{mk\text{-}sequence\text{-}Stmt}(s :: w), st) & \stackrel{\mathrm{def}}{=} & \mathsf{stmtrans}(s, st) \\
& & \mathsf{stmtrans}(\mathsf{mk\text{-}sequence\text{-}Stmt}(w), st) \\
\mathsf{stmtrans}(\mathsf{mk\text{-}if\text{-}Stmt}(b, s_1, s_2), st) & \stackrel{\mathrm{def}}{=} & \mathbf{let}\ \mathit{else} = \mathsf{newlabel}\ \mathbf{in} \\
& & \mathbf{let}\ \mathit{end} = \mathsf{newlabel}\ \mathbf{in} \\
& & \quad \mathsf{bexptrans}(b, st) \\
& & \quad \mathtt{brFalse}\ \mathit{else} \\
& & \quad \mathsf{stmtrans}(s_1, st) \\
& & \quad \mathtt{branch}\ \mathit{end} \\
& & \quad \mathit{else} : \mathsf{stmtrans}(s_2, st) \\
& & \quad \mathit{end} : \\
\mathsf{stmtrans}(\mathsf{mk\text{-}while\text{-}Stmt}(b, s), st) & \stackrel{\mathrm{def}}{=} & \mathbf{let}\ \mathit{start} = \mathsf{newlabel}\ \mathbf{in} \\
& & \mathbf{let}\ \mathit{end} = \mathsf{newlabel}\ \mathbf{in} \\
& & \quad \mathit{start} : \mathsf{bexptrans}(b, st) \\
& & \quad \mathtt{brFalse}\ \mathit{end}
\end{array}$$

$$\begin{array}{l} \mathsf{stmtrans}(s, st) \\ \texttt{branch}\ \mathit{start} \\ \mathit{end}: \end{array}$$

2.7.4 Übersetzung von Programmen

$$\mathsf{trans} : \mathsf{Prog} \to \mathsf{ZACode}$$

$$\mathsf{trans}(\mathsf{mk\text{-}Prog}(d, s, e)) \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{stmtrans}(s, st_1) \\ \mathsf{exptrans}(e, st_1) \end{array}$$

wobei $st_1 \stackrel{\text{def}}{=} \mathsf{update}(d, st_\emptyset)$

2.8 Korrektheit der Übersetzung

Wegen

$$\mathcal{M}[\![P]\!] : \mathbb{Z}^* \to \mathbb{Z} \quad \text{für P} \in \mathsf{Prog}$$

und

$$\mathcal{M}[\![P]\!] : SM \to SM \quad \text{für } P \in \mathsf{ZACode}$$

kann wieder $\mathcal{M}[\![P]\!] = \mathcal{M}[\![\mathsf{trans}(P)]\!]$ nicht erwartet werden. Wir definieren Eingabe- und Ausgabeabbildungen in und out wie bei PLE. Das Korrektheitsresultat ist dann in seiner Formulierung auch wie bei PLE.

2.9 Alternative Codeerzeugung mit „short circuit evaluation"

In PLA werden Boole'sche Werte nur im Zusammenhang mit bedingten Sprüngen verwendet; dann kann man aber auf die Auswertung von and, or und not auf der Maschinenebene verzichten und diese durch den Programm-Steuerfluß simulieren. Dabei werden dann Boole'sche Ausdrücke nur noch so weit ausgewertet wie unbedingt nötig; die Semantik der Boole'schen Operationen ändert sich von „strikt" zu „sequentiell". Von den logischen Operationen der Maschine werden dann nur noch die bedingten Sprünge verwendet, nicht jedoch mehr `and`, `or` und `not`. Für Modula-2 ist diese Auswertungsstrategie vorgeschrieben.

Die Idee ist, daß die bedingten Sprünge im Zusammenhang mit Boole'schen Ausdrücken sofort von bexptrans erzeugt werden. Hierzu bekommt bexptrans zwei weitere Parameter, nämlich die Sprungziele für **true** und **false**:

$$\mathsf{bexptrans} : \mathsf{BExp} \to \mathsf{Tab} \to \mathsf{Ide}^2 \to \mathsf{ZACode}$$

mit

$$\begin{array}{rcl} \mathsf{bexptrans}(\mathsf{mk\text{-}rel\text{-}BExp}(e_1, r, e_2), st, \mathit{ltrue}, \mathit{lfalse}) & \stackrel{\text{def}}{=} & \mathsf{exptrans}(e_1, st) \\ & & \mathsf{exptrans}(e_2, st) \\ & & \mathsf{reltrans}(r) \\ & & \texttt{brFalse}\ \mathit{lfalse} \\ & & \texttt{branch}\ \mathit{ltrue} \\ \mathsf{bexptrans}(\mathsf{mk\text{-}not\text{-}BExp}(b), st, \mathit{ltrue}, \mathit{lfalse}) & \stackrel{\text{def}}{=} & \mathsf{bexptrans}(b, st, \mathit{lfalse}, \mathit{ltrue}) \\ \mathsf{bexptrans}(\mathsf{mk\text{-}infix\text{-}BExp}(b_1, o, b_2), st, \mathit{ltrue}, \mathit{lfalse}) & \stackrel{\text{def}}{=} & \mathbf{let}\ \mathit{lsecond} = \mathsf{newlabel}\ \mathbf{in} \end{array}$$

$$
\begin{array}{l}
\textbf{if } o = \text{‘and’ } \textbf{then} \\
\quad \mathsf{bexptrans}(b_1, st, \mathit{lsecond}, \mathit{lfalse}) \\
\quad \mathit{lsecond} : \mathsf{bexptrans}(b_2, st, \mathit{ltrue}, \mathit{lfalse}) \\
\textbf{elsif } o = \text{‘or’ } \textbf{then} \\
\quad \mathsf{bexptrans}(b_1, st, \mathit{ltrue}, \mathit{lsecond}) \\
\quad \mathit{lsecond} : \mathsf{bexptrans}(b_2, st, \mathit{ltrue}, \mathit{lfalse})
\end{array}
$$

Beachte, daß durch dieses Schema häufiger redundante Sprünge auf die jeweils nächste Instruktion ausgeführt werden! Diese geben Anlaß für eine Optimierung, die man rein auf der Basis des ZA-Codes, d.h. ohne Kenntnis des Quellprogramms, durchführen kann.

Die Übersetzung der **if**- und **while**-Anweisungen muß modifiziert werden:

$$
\begin{array}{rcl}
\mathsf{stmtrans}(\mathsf{mk\text{-}if\text{-}Stmt}(b, s_1, s_2), st) & \stackrel{\text{def}}{=} & \textbf{let } \mathit{then} = \mathsf{newlabel} \textbf{ in} \\
 & & \textbf{let } \mathit{else} = \mathsf{newlabel} \textbf{ in} \\
 & & \textbf{let } \mathit{end} = \mathsf{newlabel} \textbf{ in} \\
 & & \quad \mathsf{bexptrans}(b, st, \mathit{then}, \mathit{else}) \\
 & & \quad \mathit{then} : \mathsf{stmtrans}(s_1, st) \\
 & & \quad \mathtt{branch}\ \mathit{end} \\
 & & \quad \mathit{else} : \mathsf{stmtrans}(s_2, st) \\
 & & \quad \mathit{end} : \\
\mathsf{stmtrans}(\mathsf{mk\text{-}while\text{-}Stmt}(b, s), st) & \stackrel{\text{def}}{=} & \textbf{let } \mathit{start} = \mathsf{newlabel} \textbf{ in} \\
 & & \textbf{let } \mathit{body} = \mathsf{newlabel} \textbf{ in} \\
 & & \textbf{let } \mathit{end} = \mathsf{newlabel} \textbf{ in} \\
 & & \quad \mathit{start} : \mathsf{bexptrans}(b, st, \mathit{body}, \mathit{end}) \\
 & & \quad \mathit{body} : \mathsf{stmtrans}(s, st) \\
 & & \quad \mathtt{branch}\ \mathit{start} \\
 & & \quad \mathit{end} :
\end{array}
$$

3 PL0—Programmiersprache mit Blockschachtelung

Wir erweitern jetzt PLA zu PL0 [13] bzw. MINI-PASCAL [7]; auf der Ebene der abstrakten Syntax verschwindet der Unterschied zwischen den beiden Sprachen.

PL0 kennt geschachtelte Prozeduren mit jeweils eigenen Sichtbarkeitsbereichen („scopes"), aber noch keine Prozedurparameter.

3.1 Abstrakte Syntax

Decl : Die Menge der *Deklarationen* mit den Konstruktoren:

$$\begin{array}{rcl} \mathsf{Decl} & : & \mathsf{mk\text{-}Decl(List(Const\text{-}Decl),List(Var\text{-}Decl),List(Proc\text{-}Decl))} \\ \mathsf{Const\text{-}Decl} & : & \mathsf{mk\text{-}Const\text{-}Decl(Ide,}\mathbb{Z}) \\ \mathsf{Var\text{-}Decl} & : & \mathsf{mk\text{-}Var\text{-}Decl(Ide)} \\ \mathsf{Proc\text{-}Decl} & : & \mathsf{mk\text{-}Proc\text{-}Decl(Ide,Block)} \end{array}$$

Stmt : wie bei PLA, jedoch zusätzlich

$$\mathsf{mk\text{-}call\text{-}Stmt(Ide)}$$

Block : Die Menge der *Blöcke*:

$$\mathsf{Block} \quad : \quad \mathsf{mk\text{-}Block(Decl,Stmt)}$$

Prog : Die Menge der *Programme*:

$$\mathsf{Prog} = \mathsf{Block}$$

3.2 Semantische Bereiche

Level $\stackrel{\text{def}}{=} \mathbb{N}$ die *Schachtelungsstufen*

Kind $\stackrel{\text{def}}{=} \{\mathsf{const}, \mathsf{var}, \mathsf{proc}\}$

Env $\stackrel{\text{def}}{=} \mathsf{Ide} \to \mathsf{Kind} \times \mathsf{Level} \times (\mathbb{Z} \cup \mathsf{Loc} \cup \mathsf{Block})$

Das Vorkommen von Block, also eines syntaktischen Bereichs in einer Komponente von Env mag hier befremdlich vorkommen; dies hat mit der Möglichkeit des rekursiven Aufrufs von Prozeduren zu tun; s. nächster Abschnitt.

3.3 Semantik von Deklarationen

Wegen der geschachtelten Sichtbarkeitsbereiche wird die Semantik von Deklarationen jetzt etwas komplizierter; ebenso verkompliziert sich die Beschreibung der statischen Semantik. So muß etwa die Verschiedenheit der Bezeichner nur auf der Stufe der Deklarationen eines einzelnen Blocks gelten; Deklarationen aus umgebenden Blöcken dürfen durchaus „verschattet" werden. Um dies realisieren zu können, haben wir bereits die Umgebungen um eine Komponente Level erweitert. Aber auch die Zuordnung von Speicherplätzen zu Variablen wird komplizierter: Wegen der Möglichkeit des rekursiven Aufrufs von Prozeduren kann einer Variablen nun nicht mehr dauerhaft ein fester Speicherplatz zugewiesen werden; vielmehr muß beim Betreten eines jeden Blocks eine *Allokation* der in diesem Block deklarierten Variablen erfolgen. Da Blöcke durch die Prozeduraufrufe *dynamisch*, d.h. in Abhängigkeit von der Programmausführung mit bestimmten Daten, betreten werden können, während wir hier bei der Semantikbeschreibung nur *statische Informationen*, nämlich die abstrakte Syntax des Programms zur Verfügung haben, ist es die einfachste Lösung für dieses Dilemma, die zu einem Prozedurblock gehörende abstrakte Syntax in die Umgebung einzutragen.

$\mathcal{D}$ die Semantik von Deklarationen: $\mathcal{D}$: $\mathsf{Decl} \to \mathsf{Level} \to \mathsf{Env} \to \mathsf{Env}$

$$\mathcal{D}[\![\mathsf{mk\text{-}Decl}(d_c, d_v, d_p)]\!] l\rho \stackrel{\text{def}}{=} \mathcal{D}_p[\![d_p]\!] l(\mathcal{D}_v[\![d_v]\!] l(\mathcal{D}_c[\![d_c]\!] l\rho))$$

$$\mathcal{D}_v[\![\mathsf{nil}]\!] l\rho \stackrel{\text{def}}{=} \mathcal{D}_c[\![\mathsf{nil}]\!] l\rho \stackrel{\text{def}}{=} \mathcal{D}_p[\![\mathsf{nil}]\!] l\rho \stackrel{\text{def}}{=} \rho$$

$$\mathcal{D}_c[\![\mathsf{mk\text{-}Const\text{-}Decl}(i, z) :: w]\!] l\rho \stackrel{\text{def}}{=} \mathcal{D}_c[\![w]\!] l\rho[i \mapsto (\mathsf{const}, l, z)] \quad (1)$$

$$\mathcal{D}_v[\![\mathsf{mk\text{-}Var\text{-}Decl}(i) :: w]\!] l\rho \stackrel{\text{def}}{=} \mathcal{D}_v[\![w]\!] l\rho[i \mapsto (\mathsf{var}, l, \mathsf{new}(\rho))] \quad (2)$$

$$\mathcal{D}_p[\![\mathsf{mk\text{-}Proc\text{-}Decl}(i, b) :: w]\!] l\rho \stackrel{\text{def}}{=} \mathcal{D}_p[\![w]\!] l\rho[i \mapsto (\mathsf{proc}, l, b)] \quad (3)$$

3.4 Statische Semantik

1. Verschiedenheit der Bezeichner: In den Gleichungen 1–3 von Abschnitt 3.3 muß jeweils auf der linken Seite gelten
   $$\rho(i) \neq (-, l, -)$$
   Dabei steht „–“ für einen beliebigen Wert. Die Forderung bedeutet, daß der Bezeichner i in ρ noch nicht mit der aktuellen Schachtelungstiefe l deklariert sein darf; über andere Deklarationen (d.h. also mit kleinerer Schachtelungstiefe) ist nichts ausgesagt.

2. Deklarationszwang und richtiger syntaktischer Typ

   Bei PLE und PLA hatten wir es nur mit einer einzigen Umgebung zu tun; für PL0 müssen wir an jeder Stelle die jeweils *relevante Umgebung* betrachten. Was die relevante Umgebung ist, ergibt sich aus der Anwendung der semantischen Funktionen: eine Umgebung ρ ist relevant, wenn zur Auswertung von $\mathcal{M}[\![p]\!]$ eine Semantikfunktion mit ρ als Argument aufgerufen wird. Wir formulieren einfach wie folgt:

   Für $p = \mathsf{mk\text{-}Block}(d, b)$ und $i \in \mathsf{Ide}$ muß gelten:

   (a) (analog PLE) In $\mathcal{E}[\![\mathsf{mk\text{-}Const\text{-}Exp}(i)]\!]\rho$ muß gelten $\rho(i) = (\mathsf{const}, l, z)$ für ein $l \in \mathsf{Level}$ und ein $z \in \mathbb{Z}$.
   (b) (analog PLE) In $\mathcal{E}[\![\mathsf{mk\text{-}Var\text{-}Exp}(i)]\!]\rho$ muß gelten $\rho(i) = (\mathsf{var}, l, \alpha)$ für ein $l \in \mathsf{Level}$ und ein $\alpha \in \mathsf{Loc}$.
   (c) (analog PLA) In $\mathcal{S}[\![\mathsf{mk\text{-}assign\text{-}Stmt}(i, e)]\!]\rho$ muß gelten $\rho(i) = (\mathsf{var}, l, \alpha)$ für ein $l \in \mathsf{Level}$ und ein $\alpha \in \mathsf{Loc}$.
   (d) In $\mathcal{S}[\![\mathsf{mk\text{-}call\text{-}Stmt}(i)]\!]\rho$ muß gelten $\rho(i) = (\mathsf{proc}, l, \beta)$ für ein $l \in \mathsf{Level}$ und ein $\beta \in \mathsf{Stat}$.

3.5 Semantische Funktionen

$\mathcal{S}$: $\mathsf{Stmt} \to \mathsf{Env} \to \mathsf{Stat}$ wie bei PLA (beachte $\mathsf{Stat} = \mathsf{Store} \to \mathsf{Store}$) mit der Erweiterung

$$\mathcal{S}[\![\mathsf{mk\text{-}call\text{-}Stmt}(i)]\!]\rho\sigma \stackrel{\text{def}}{=} \mathbf{let}\ (\mathsf{proc}, l, b) = \rho(i)\ \mathbf{in}\ \mathcal{L}[\![b]\!](l+1)\rho\sigma$$

Dabei ist $\mathcal{L}$ die Semantik von *Blöcken*, die wir im Anschluß definieren werden. Wir haben damit übrigens nicht Pascal beschrieben, denn bei der Ermittlung der Blocksemantik von b müßte die Umgebung *der Deklarationsstelle* herangezogen werden, nicht die der Aufrufstelle! („static scope“: die syntaktische Schachtelungsstruktur entscheidet über die Sichtbarkeit von Bezeichnern.) Für die Blocksemantik des Rumpfs wird die Schachtelungstiefe um eins erhöht, was zur Folge hat, daß die in der Prozedur deklarierten lokalen Konstanten, Variablen und Prozeduren in den nächsten Sichtbarkeitsbereich gelangen.

$\mathcal{L}$: Block $\to$ Level $\to$ Env $\to$ Stat die *Semantik von Blöcken*:

$$\mathcal{L}[\![\text{mk-Block}(d,s)]\!]l\rho\sigma \stackrel{\text{def}}{=} \mathcal{S}[\![s]\!](\mathcal{D}[\![d]\!]l\rho)\sigma'$$

wobei

$$\sigma'(\alpha) = \begin{cases} 0 & \text{für die in } d \text{ neu angelegten Speicherplätze} \\ \sigma(\alpha) & \text{sonst} \end{cases}$$

Die Deklarationssemantik $\mathcal{D}$ legt für die in diesem Block neu deklarierten Variablen neue Speicherplätze an. Da wir die Blocksemantik $\mathcal{L}$ von Prozeduren erst bei der Berechnung der Semantik von Prozeduraufrufen auswerten, gibt es wirklich für jeden Prozeduraufruf neue Speicherplätze für die lokalen Variablen. Über die Notwendigkeit der hier durchgeführten Initialisierung lokaler Variablen ließe sich sicher streiten; in einigen Programmiersprachen (z.B. C, PL/I) ist sie üblich, in anderen nicht (z.B. Pascal, Modula-2). Besteht man wie hier auf einer (automatischen) Initialisierung von Variablen, so muß in der Implementierung bei jedem Aufruf im Rahmen des sogenannten *Prozedur-Prologs* Code für diese Initialisierung ausgeführt werden, was natürlich Leistung kostet. Verzichtet man dagegen auf die Initialisierung, so hätte man es im Grunde genommen als einen Laufzeitfehler zu betrachten, wenn ein Programm auf eine nicht initialisierte lokale Variable zugreift. Will man dies in der Implementierung korrekt widerspiegeln, so würde dies noch teurer als die einfache Initialisierung: Wir müßten einen Datenwert opfern, den wir zur Charakterisierung des nicht initialisierten Zustands verwenden, jede Variable mit diesem speziellen Wert initialisieren und jeden lesende Variablenzugriff mit einem vorherigen Test auf diesen Wert abschotten. Wegen der hohen Effizienzverluste wird dies jedoch nie implementiert. Wir wollen hier keine Lücken in der Definition lassen und führen deshalb die Initialisierung durch.

$\mathcal{M}$ die Semantik von Programmen: $\mathcal{M}$: Prog $\to$ Stat , mit

$$\mathcal{M}[\![\text{mk-Block}(d,s)]\!] \stackrel{\text{def}}{=} \mathcal{L}[\![\text{mk-Block}(d,s)]\!]0\rho_\emptyset$$

3.6 Zwischencode Z0 für Blöcke

Wir gehen von dem Zwischencode ZA aus. Wegen der Blockschachtelung brauchen wir jetzt jedoch eine kompliziertere Anlage des Stapels, denn eine Nachbildung der Mechanismen bei unserer Semantikdefinition scheidet aus. (Hier müßte man bei jedem Prozeduraufruf im Prinzip den Prozedurrumpf während der Laufzeit neu compilieren, ein unmögliches Verfahren.) Wir müssen bei jedem Betreten einer Prozedur einen neuen *Aktivierungsblock* für die Rücksprungadresse und die lokalen Variablen anlegen können, der beim Verlassen der Prozedur wieder beseitigt werden muß. Außerdem müssen wir in jedem Block außer auf die darin deklarierten (*„lokalen“*) Variablen auch auf die Variablen aller umgebenden Sichtbarkeitsbereiches zugreifen können. Wir richten deshalb in der abstrakten Maschine ein sogenanntes *„Display“* ein[3], welches Anfangsadressen von Aktivierungsblöcken enthält. Wegen des Entfernens von Aktivierungsblöcken brauchen wir jetzt eine Operation zum Verkürzen eines Stapels: Für $d = d_1 \ldots d_n$ und $m < n$ sei $d|_m \stackrel{\text{def}}{=} d_1 \ldots d_m$.

Die Aktivierungsblöcke heißen englisch auch *„frames“*; ihre Behandlung auf der Maschinenebene, speziell das Entfernen eines Aktivierungsblocks beim Verlassen einer Prozedur, kann erleichtert werden, wenn eine zusätzliche Komponente FP *(frame pointer)* eingeführt wird.

Zustandsmenge: $SM \stackrel{\text{def}}{=} IP \times S \times FP \times D$, wobei IP, S wie bei PLA und

$$FP \stackrel{\text{def}}{=} \mathbb{N} \quad \textit{(Frame Pointer)}$$
$$D \stackrel{\text{def}}{=} \mathbb{N}^* \quad \textit{(Display)}$$

[3] Genauer gesagt, handelt es sich um ein *globales Display*

Befehlsvorrat: [4]

$$\Gamma_{\mathtt{Z0}} \stackrel{\text{def}}{=} \Gamma_{\mathtt{ZA}} \cup \{\mathtt{call}\ l, \mathtt{enter}\ i, z, \mathtt{leave}\ i, \mathtt{ret}, \mathtt{pshDsp}\ i, o, \mathtt{pshFP}\ o \mid l \in \mathsf{Ide}, i, z, o \in \mathbb{N}\}$$

Der Befehl $\mathtt{call}\ l$ überträgt die Kontrolle an die Prozedur l und hinterläßt die Rücksprungadresse auf dem Stapel. $\mathtt{enter}\ i, z$ erzeugt einen neuen Aktivierungsblock, der zunächst den augenblicklichen Wert von FP („dynamischer Verweis"), dann den augenblicklichen Wert von $d(i)$ und danach z mit $\mathtt{0}$ initialisierte lokale Variablen enthält. Im Display D wird an der i-ten Stelle die Anfangsadresse des neuen Aktivierungsblocks eingetragen, damit evtl. eingeschachtelte Blöcke auf die lokalen Variablen zugreifen können. FP erhält die Adresse des neuen Aktivierungsblocks. Die zu `enter` umgekehrte Wirkung hat der Befehl $\mathtt{leave}\ i$: durch Zugriff auf FP werden die lokalen Variablen vom Stapel entfernt, der alte Inhalt von $d(i)$ und der alte Inhalt von FP wiederhergestellt. `ret` entfernt die Rücksprungadresse vom Stapel und setzt die Verarbeitung an dieser Stelle fort.

$\mathtt{pshDsp}\ i, o$ lädt die absolute Stapeladresse des Objekts mit relativer Adresse o im Aktivierungsblock mit Schachtelungsnummer i auf den Stapel. $\mathtt{pshFP}\ o$ lädt die absolute Adresse des Objekts mit relativer Adresse o im obersten Aktivierungsblock auf den Stapel.

Befehlssemantik: Jeder Befehl $\gamma \in \Gamma_{\mathtt{ZA}}$ bestimmt eine partielle Zustandstransformation $\mathcal{C}[\![\gamma]\!] \in SM \to SM$, wobei die ZA-Befehle wie bei PLA definiert seien mit der Identität auf den dort nicht benutzten D- und FP-Komponenten. Definiere dann

$$\begin{aligned}
\mathcal{C}[\![\mathtt{call}\ l]\!](m,s,f,d) &\stackrel{\text{def}}{=} (i_l, s.m+1, f, d) \\
&\phantom{\stackrel{\text{def}}{=}} \text{wobei } i_l \text{ die Nummer der Instruktion mit Marke } l \text{ ist} \\
\mathcal{C}[\![\mathtt{enter}\ i,z]\!](m,s,f,d) &\stackrel{\text{def}}{=} (m+1, s.f.d(i).\overbrace{0\ldots0}^{z}, \mathsf{len}(s)+1, d[i \mapsto \mathsf{len}(s)+1]) \\
\mathcal{C}[\![\mathtt{leave}\ i]\!](m,s,f,d) &\stackrel{\text{def}}{=} (m+1, s|_{f-1}, s(f), d[i \mapsto s(f+1)]) \\
\mathcal{C}[\![\mathtt{ret}]\!](m,s.r,f,d) &\stackrel{\text{def}}{=} (r,s,f,d) \\
\mathcal{C}[\![\mathtt{pshDsp}\ i,o]\!](m,s,f,d) &\stackrel{\text{def}}{=} (m+1, s.d(i)+o, f, d) \\
\mathcal{C}[\![\mathtt{pshFP}\ o]\!](m,s,f,d) &\stackrel{\text{def}}{=} (m+1, s.f+o, f, d)
\end{aligned}$$

Programme ($\mathsf{Z0Code}$), Einzelschrittsemantik und Iterationssemantik sind analog zu PLA definiert.

3.7 Übersetzung von PL0 nach Z0

3.7.1 Symboltabelle

Gegenüber PLE wird die Symboltabelle jetzt zunächst um einen Eintrag für Prozeduren erweitert; Attribut von Prozeduren ist die Deklarationstiefe *level*. Variablen erhalten ebenfalls die Deklarationstiefe als weiteres Attribut. Auch die Funktion update erhält die augenblickliche Schachtelungstiefe als zusätzliches Argument. Weiterhin sei jetzt

$$\mathsf{newloc} : \mathsf{Level} \times \mathsf{Tab} \to \mathsf{Loc}$$

eine Funktion, die jeweils den niedrigsten Speicherplatz zurückgibt, der in der Symboltabelle noch nicht einer Variablen eines bestimmten Deklarationsniveaus zugeordnet wurde und

$$\mathsf{size} : \mathsf{Block} \to \mathbb{N}$$

[4] Die Befehle `enter` und `leave` sind in Wirklichkeit keine D-Code-Befehle nach [3]; dort wird das Betreten und Verlassen von Prozeduren z.T. durch Pseudo-Instruktionen (`.PROC`, `.ENDP`) beschrieben.

eine Funktion, welche die Anzahl der in einem Block deklarierten lokalen Variablen angibt. (In einem Compiler wird man diese Zahl als Attribut eines Blocks in der Symboltabelle vermerken, statt sie bei jeder Gelegenheit neu zu berechnen.) Dann ist

$$\mathsf{Tab} \stackrel{\text{def}}{=} \mathsf{Ide} \to (\{\mathsf{const}\} \times \mathbb{Z}) \cup (\{\mathsf{var}\} \times \mathsf{Level} \times \mathsf{Loc}) \cup (\mathsf{proc} \times \mathsf{Level}) .$$

und

$$\mathsf{update} : \mathsf{Decl} \to \mathsf{Level} \to \mathsf{Tab} \to \mathsf{Tab}$$

definiert durch

$$\begin{aligned}
\mathsf{update}(\mathsf{mk\text{-}Decl}(d_c, d_v, d_p), l, st) &\stackrel{\text{def}}{=} \mathsf{p\text{-}update}(d_p, l, \mathsf{v\text{-}update}(d_v, l, \mathsf{c\text{-}update}(d_c, l, st))) \\
\mathsf{p\text{-}update}(\mathsf{nil}, l, st) &\stackrel{\text{def}}{=} \mathsf{v\text{-}update}(\mathsf{nil}, l, st) \stackrel{\text{def}}{=} \mathsf{c\text{-}update}(\mathsf{nil}, l, st) \stackrel{\text{def}}{=} st \\
\mathsf{c\text{-}update}(\mathsf{mk\text{-}Const\text{-}Decl}(i, z) :: w, l, st) &\stackrel{\text{def}}{=} \mathsf{c\text{-}update}(w, l, st[i \mapsto (\mathsf{const}, z)]) \\
\mathsf{v\text{-}update}(\mathsf{mk\text{-}Var\text{-}Decl}(i) :: w, l, st) &\stackrel{\text{def}}{=} \mathsf{v\text{-}update}(w, l, st[i \mapsto (\mathsf{var}, l, \mathsf{newloc}(l, st))]) \\
\mathsf{p\text{-}update}(\mathsf{mk\text{-}Proc\text{-}Decl}(i, b) :: w, l, st) &\stackrel{\text{def}}{=} \mathsf{p\text{-}update}(w, l, st[i \mapsto (\mathsf{proc}, l)])
\end{aligned}$$

3.7.2 Übersetzung von Deklarationen

Wegen der eingeschachtelten Prozeduren brauchen wir jetzt erstmals eine Code-Erzeugung für Deklarationen: Beim Lesen einer Prozedurdeklaration wird Code für deren Rumpf erzeugt. Im Rahmen dieses Vorgangs passen wir dann auch die Symboltabelle an. Im Gegensatz zum Wirth'schen Ansatz [13] gehen wir nicht davon aus, daß der Einsprungspunkt des ganzen Programms die erste Zwischencode-Instruktion ist (was dazu führt, daß der Code für die eingeschachtelten Prozeduren übersprungen werden muß), sondern wählen statt dessen eine mit *main* markierte Instruktion. Eine neue Übersetzungsfunktion blocktrans übersetzt einen Block, indem zunächst die eingeschachtelten Prozedurdeklarationen übersetzt werden und dann hinter einer übergebenen Einsprungsmarke der Rumpf des Blocks.

Definiere

$$\mathsf{decltrans} : \mathsf{Decl} \to \mathsf{Level} \to \mathsf{Tab} \to \mathsf{Z0Code}$$

durch

$$\begin{aligned}
\mathsf{decltrans}(\mathsf{mk\text{-}Decl}(D_c, D_v, D_p), l, st) &\stackrel{\text{def}}{=} \mathsf{p\text{-}decltrans}(D_p, l, st) \\
\mathsf{p\text{-}decltrans}(\mathsf{nil}, l, st) &\stackrel{\text{def}}{=} \mathsf{nil} \\
\mathsf{p\text{-}decltrans}(\mathsf{mk\text{-}Proc\text{-}Decl}(i, b) :: w, l, st) &\stackrel{\text{def}}{=} \mathsf{blocktrans}(b, l+1, i, st[i \mapsto (\mathsf{proc}, l)]) \\
&\qquad \mathsf{p\text{-}decltrans}(w, l, st[i \mapsto (\mathsf{proc}, l)])
\end{aligned}$$

Beachte die Erhöhung der Schachtelungstiefe!

Scheinbar haben wir hier eine Aufgabe dreifach erledigt, nämlich das Eintragen von i als Prozedur in die Symboltabelle. Dieses wird nämlich außer an den beiden hier vorkommenden Stellen auch noch von p-update erledigt. Ein echter Compiler wird (hauptsächlich wegen der Anlage der Symboltabelle als einer globalen Variablen) die Eintragung natürlich nur einmal vornehmen; in der hier vorgenommenen funktionalen Beschreibung wird jedoch deutlich, daß eine Prozedurdeklaration an drei Stellen Wirkung zeigt:

1. in den evtl. eingeschachtelten Prozedurdeklarationen (blocktrans(...)),
2. in den nachgeschalteten Prozedurdeklarationen derselben Deklarationsliste (p-decltrans(...)) und
3. im Rumpf des zugehörigen Blocks (p-update(...)).

3.7.3 Übersetzung von Variablen

Die Übersetzung von Variablen, die bislang innerhalb der Ausdrucksübersetzung exptrans und der Übersetzung der Zuweisung stmtrans(mk-assign-stmt(...)) besorgt wurde, wird jetzt so kompliziert, daß es sich lohnt, sie separat zu betrachten.

$$\mathsf{vartrans} : \mathsf{Ide} \to \mathsf{Level} \to \mathsf{Tab} \to \mathsf{Z0Code}$$

$$\mathsf{vartrans}(i, l, st) \stackrel{\text{def}}{=} \begin{cases} \texttt{pshAdr}\ n+3 & \text{falls } st(i) = (\mathsf{var}, 0, n) \\ \texttt{pshFP}\ o+2 & \text{falls } st(i) = (\mathsf{var}, l, o) \\ \texttt{pshDsp}\ v, o+2 & \text{falls } st(i) = (\mathsf{var}, v, o) \wedge 0 \neq v \neq l \end{cases}$$

Der Grund, warum wir bei den globalen Variablen auf die n-te Variable mit Adresse $n+3$ zugreifen, ist der, daß wir auch das Hauptprogramm jetzt wie einen normalen Block behandeln wollen. Dann sind die ersten drei Variablenplätze aber für Rücksprungadresse, dynamischen Verweis und geretteten Display-Wert reserviert. Die Erhöhung des vorgegebenen Offsets o um 2 bei `pshFP` und `pshDsp` geschieht, weil die erste lokale Variable eines Aktivierungsblocks die relative Adresse 0 hat; zum Zugriff müssen noch der dynamische Verweis und der gerettete Display-Eintrag übersprungen werden.

3.7.4 Übersetzung von Ausdrücken

Gegenüber PLA hat jetzt exptrans den Typ

$$\mathsf{exptrans} : \mathsf{Exp} \to \mathsf{Level} \to \mathsf{Tab} \to \mathsf{Z0Code}$$

Das zusätzliche Argument $l \in$ Level (Schachtelungstiefe) wird meistens ignoriert mit der Ausnahme:

$$\mathsf{exptrans}(\mathsf{mk\text{-}Var\text{-}Exp}(i), l, st) \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{vartrans}(i, l, st) \\ \texttt{deref} \end{array}$$

3.7.5 Übersetzung von Anweisungen

Definiere

$$\mathsf{stmtrans} : \mathsf{Stmt} \to \mathsf{Level} \to \mathsf{Tab} \to \mathsf{Z0Code}$$

analog zu PLA mit der Änderung

$$\mathsf{stmtrans}(\mathsf{mk\text{-}assign\text{-}Stmt}(i, e), l, st) = \begin{array}{l} \mathsf{exptrans}(e, l, st) \\ \mathsf{vartrans}(i, l, st) \\ \texttt{assign} \end{array}$$

und der Erweiterung

$$\mathsf{stmtrans}(\mathsf{mk\text{-}call\text{-}Stmt}(i), l, st) = \texttt{call}\ i$$

3.7.6 Übersetzung von Blöcken

$$\mathsf{blocktrans} : \mathsf{Block} \to \mathsf{Level} \to \mathsf{Ide} \to \mathsf{Tab} \to \mathsf{Z0Code}$$

$$\begin{array}{lcl} \mathsf{blocktrans}(\mathsf{mk\text{-}Block}(d,s),l,i,st) & \stackrel{\text{def}}{=} & \mathsf{decltrans}(d,l,st) \\ & & i : \mathtt{enter}\ l, \mathsf{size}(d) \\ & & \mathsf{stmtrans}(s,l,\mathsf{update}(d,l,st)) \\ & & \mathtt{leave}\ l \\ & & \mathtt{ret} \end{array}$$

3.7.7 Übersetzung von Programmen

$$\mathsf{trans} : \mathsf{Prog} \to \mathsf{Z0Code}$$

$$\mathsf{trans}(\mathsf{mk\text{-}Block}(d,s)) \stackrel{\text{def}}{=} \mathsf{blocktrans}(\mathsf{mk\text{-}Block}(d,s), 0, \mathit{main}, st_{\emptyset})$$

3.8 Korrektheit der Übersetzung

In PLA hatten wir es noch erzwungen, daß die Semantik eines Programms eine ganze Zahl ist. Das läßt sich in PL0 nur noch mit Mühe konsequent fortführen, so daß wir die Semantik eines Programms als Speichertransformation definiert haben. Jetzt wird aber die Korrektheit etwas schwieriger zu formulieren; wir müßten verlangen, daß nach Ausführung des Programms jede Variable des Hauptprogramms den Wert besitzt, der sich aus der denotationellen Semantik ergibt. Das Problem ist aber, daß vor dem Start des Programms der Aktivierungsblock des Hauptprogramms gar nicht existiert; dieser wird erst durch die `enter`-Instruktion des Hauptprogramms erzeugt! Eine „Eingabe" in den Stapel, wie bislang durchgeführt, ist also gar nicht möglich. Umgekehrt wird vor dem Verlassen des Hauptprogramms durch die `ret`-Instruktion sogar der Aktivierungsblock des Hauptprogramms mittels `leave` entfernt, sodaß sozusagen alle Beweismittel vernichtet sind. Für PL0 müßten wir eine von zwei Maßnahmen treffen:

1. Ein- und Ausgabeanweisungen einführen und die Korrektheit bezüglich irgendwelcher konkreter Ein/Ausgabe-Transformationen nachweisen. Das kann durchaus kompliziert werden, da Ein- bzw. Ausgabeanweisungen dann auch in Schleifen vorkommen können etc.
2. Umgebungen und Speicher der denotationellen Semantik systematisch in Symboltabellen und Z0-Maschinenkonfigurationen und zurück übersetzen und dann jeweils induktiv die Korrektheit der Übersetzung einzelner Konstrukte beweisen.

In dieser Vorlesung verzichten wir allerdings darauf, zumal wir die Korrektheit sowieso nie formal beweisen. In jedem Fall ist die Startkonfiguration der Z0-Maschine gegeben durch

$$(i_{main}, 0, 0, \varepsilon)$$

wobei i_{main} die Adresse der Instruktion mit Marke *main* ist und ε ein leeres Display. Der Anfangsstapel enthält eine 0, die durch den `ret`-Befehl am Ende des Programms als Rücksprungadresse interpretiert wird und damit den Programmlauf beendet.

3.9 Beispiel

Wir zeigen ein PL0-Programm in konkreter Syntax und das daraus resultierende Z0-Maschinenprogramm. Dieses wurde von einem mit `yacc` und Modula-2 erzeugten Compiler generiert. In Abweichung von unserer abstrakten Syntax enthält dieses Programm auch eine **IF**-Anweisung ohne **ELSE**-Zweig, was sich jedoch trivial handhaben läßt. (In unserem Übersetzungsschema führt dies zu einem redundanten Sprung auf die Marke `if4`.

PROGRAM *auf46*;

CONST *a* = 2;

VAR *i*,*j*: *Integer*;

PROCEDURE *p*;
VAR *i*: *Integer*;

BEGIN
    i := *j*;
    WHILE *i* > 0 **DO BEGIN**
        j := *j*+*a*;
        i := *i*–1;
        IF *j* > 20 **THEN**
            IF *i* <> 3 **THEN**
                j := *j* **DIV** 3
            ELSE *j* := *j*–1
    END;
END;

BEGIN
    WHILE *i* >= 0 **DO BEGIN**
        p;
        i := *i*–1
    END
END.

```
p                    :
        enter   1,1
        pshAdr  4       ; j
        deref
        pshFP   2       ; i
        assign
while1:
        pshFP   2       ; i
        deref
        pshLit  0
        relGT
        brFalse while2
        pshAdr  4       ; j
        deref
        pshLit  2       ; a
        add
        pshAdr  4       ; j
        assign
        pshFP   2       ; i
        deref
        pshLit  1
        sub
        pshFP   2       ; i
        assign
```

```
        pshAdr  4       ; j
        deref
        pshLit  2
        relGT
        brFalse if3
        pshFP   2       ; i
        deref
        pshLit  3
        relNE
        brFalse if5
        pshAdr  4       ; j
        deref
        pshLit  3
        div
        pshAdr  4       ; j
        assign
        branch  if6
if5:
        pshAdr  4       ; j
        deref
        pshLit  1
        sub
        pshAdr  4       ; j
        assign
if6:
        branch  if4
if3:
if4:

        branch  while1
while2:


        leave   1
        ret

main:
        enter   0,2
while7:
        pshAdr  3       ; i
        deref
        pshLit  0
        relGE
        brFalse while8
        call    p
        pshAdr  3       ; i
        deref
        pshLit  1
        sub
        pshAdr  3       ; i
        assign
```

```
        branch  while7
while8:
        leave   0
        ret
```

Ein weiteres Beispiel soll die Implementierung der Variablenzugriffe über das Display illustrieren:

PROGRAM *scopes*;

VAR *a*,*b*,*c*,*d*,*e*: *Integer*;

PROCEDURE *p*;

(∗ *Wenn wir bis zu dieser Stelle gelesen haben, sieht die Symboltabelle so aus:*

Name	*Art*	*Level*	*Offset*
a	var	*0*	*0*
b	var	*0*	*1*
c	var	*0*	*2*
d	var	*0*	*3*
e	var	*0*	*4*
p	proc	*0*	

∗)

VAR *a*,*b*,*d*: *Integer*;

PROCEDURE *q*;

(∗ *Jetzt sieht die Symboltabelle so aus:*

Name	*Art*	*Level*	*Offset*
a	var	*1*	*0*
b	var	*1*	*1*
c	var	*0*	*2*
d	var	*1*	*2*
e	var	*0*	*4*
p	proc	*0*	
q	proc	*1*	

∗)

VAR *c*: *Integer*;

(∗ *und jetzt so:*

Name	*Art*	*Level*	*Offset*
a	var	*1*	*0*
b	var	*1*	*1*
c	var	*2*	*0*
d	var	*1*	*2*
e	var	*0*	*4*
p	proc	*0*	
q	proc	*1*	

∗)

BEGIN (∗ *q* ∗)
c := *d*;
e := *c* + 1;
IF *c*=0 **THEN** *p*
END; (∗ *q* ∗)

BEGIN (∗ *p* ∗)
c := *e*;
d := *c*;
q
END; (∗ *p* ∗)

BEGIN (∗ *Hauptprogramm* ∗)
p
END.

```
q                         :
        enter   2,1
        pshDsp  1,4      ; d
        deref
        pshFP   2        ; c
        assign
        pshFP   2        ; c
        deref
        pshLit  1
        add
        pshAdr  7        ; e
        assign

        pshFP   2        ; c
        deref
        pshLit  0
        relEQ
        brFalse if1
        call    p
        branch  if2
if1:
if2:

        leave   2
```

```
        ret
p                       :
        enter   1,3
        pshAdr  7       ; e
        deref
        pshAdr  5       ; c
        assign
        pshAdr  5       ; c
        deref
        pshFP   4       ; d
        assign

        call    q

        leave   1
        ret

main:
        enter   0,5
        call    p
        leave   0
        ret
```

Literatur

[1] Elfriede Fehr. *Semantik von Programmiersprachen*. Springer, 1989.

[2] John Gough. The D-Code assembler DASM. Berichte des Wilhelm-Schickard-Instituts, Universität Tübingen, 1991.

[3] John Gough. The D-Code compiler front-end GP2D. Berichte des Wilhelm-Schickard-Instituts, Universität Tübingen, 1991.

[4] Elisabeth Heyne. Compilererzeugung mit Hilfe algebraischer Spezifikationen. Diplomarbeit, RWTH Aachen, 1984.

[5] Klaus Indermark. Functional compiler description. *Banach Center Publications*, 21:257–275, 1988.

[6] Herbert Klaeren. A constructive method for abstract algebraic software specification. *Theoretical Computer Science*, 30:139–204, 1984.

[7] Herbert Klaeren. *Vom Problem zum Programm — Eine Einführung in die Informatik*. Teubner Verlag, 1990.

[8] Herbert Klaeren. Using `yacc` with Modula-2. Berichte des Wilhelm-Schickard-Instituts, Universität Tübingen, 1991.

[9] Herbert Klaeren and Heiko Petzsch. Algebraic software specification and compiler generation—a case study. Schriften zur Informatik und angewandten Mathematik 68, RWTH Aachen, 1981.

[10] Herbert Klaeren and Heiko Petzsch. The development of an interpreter by means of abstract algebraic software specifications. In *Int. Colloquium on Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 335–346. Springer Verlag, 1981.

[11] Herbert Klaeren and Peter Thiemann. A clean Modula-2 interface to abstract data types. *Structured Programming*, 11:69–77, 1990.

[12] Niklaus Wirth. Personal communication. Zürich, 1990.

[13] Niklaus Wirth. *Compilerbau*. Teubner, 1981.