MATHEMATICAL PROBLEMS IN COMPUTATION THEORY
BANACH CENTER PUBLICATIONS, VOLUME 21
PWN - POLISH SCIENTIFIC PUBLISHERS
WARSAW 1988

FUNCTIONAL COMPILER DESCRIPTION

K. INDERMARK

Lehrstuhl für Informatik II, RWTH Aachen, Bundesrepublik Deutschland

Introduction

We shall give a complete definition of a compiler front-end using structural-recursive functions with parameters. For that purpose we choose the sample language PL/O of Wirth's book on compiler construction as source language and an appropriate stack machine code as target language, slightly modified from [9]. This example is non-trivial in so far as PL/O allows nested recursive procedure declarations. Therefore, the corresponding runtime storage management has to deal with static link chains or similar techniques.

Defining the translation by structural-recursive functions with parameters has several advantages: it yields a succint, modular, and easily readable front-end description; inductive correctness proofs are possible, though not yet carried out; a rapid compiler prototype can be obtained using PROLOG or LISP because these languages permit recursive function definitions as programs.

Of course, our case study suggests more, namely taking structural-recursive functions with parameters as a functional compiler description language. Since we do not allow arbitrary recursion but only a sort of primitive recursion, a more efficient implementation than the usual stack mechanism should be possible. We hope to develop these ideas in further research.

The paper is organized as follows. In Section 1 we define our source language PL/O by giving its abstract syntax and denotational semantics. We assume that parsing has been done and we can therefore proceed by structural induction on the abstract syntax tree. This technique is not only used for defining the semantics but also for the translation.

The target language is presented in Section 2. It consists of code for a suitable stack machine. Again, we formally define syntax and semantics of its instruction set and machine programs. In particular, this includes a formal

treatment of activation records and their handling on the procedure stack by static link chains.

The final Section 3 contains the translation mapping defined by structural recursion, i.e., by much the same technique as already used in the denotational definition of PL/O. However, the translation mapping turns out to be more complex because the mathematical comfort of infinite objects, e.g. procedure meanings in environments, is no longer possible : symbol tables store only a finite amount of information for procedure identifiers.

We have tried to develop a complete formal definition of a non-trivial compiler front-end. A clear separation of the translation phase from lexical and syntactic analysis motivates the use of abstract syntax and thereby permits inductive techniques. Despite our claim of formal completeness we made an effort to keep the formalism as simple as possible. For our purpose, it suffices to use partial functions over sets instead of continuous functions over complete partial orders. In particular, no domain theory is required. For the same reason, we decided not to enlarge the translation by a formal error handling. This simplification when compared with the treatment in Milne's and Strachey's book [6] is of course due to the structure of our source language PL/O. Nevertheless, we believe that translating PL/O-programs into stack code is an essential task of compiling high-level programming languages.

1. THE SOURCE LANGUAGE PL/O

To illustrate our technique for defining a language translation by means of structural-recursive functions with parameters we choose Wirth's model language PL/O [9]. One reason for this choice is that Wirth gives already in his book a complete definition of the translation into stack code using PASCAL. We hope to demonstrate that our functional definition has the above mentioned advantages. However, what remains to be shown is that our class of recursive function definitions permits an efficient implementation.

PL/O is an imperative PASCAL-like language which abstracts from data types and data structures. On the other hand, it is block-structured and contains nested recursive procedure declarations. Hence, for static scope semantics the corresponding stack code has to deal with static link chains.

1.1. Abstract syntax of PL/O. Using abstract syntax means that syntactic objects are regarded as elements of a free many-sorted algebra, not as symbol strings. This allows the use of structural induction (or recursion) for defining semantics and translation. For simplicity we do not distinguish between an object and its denotation whenever possible.

Int: z (* z denotes an integer *)
Ide: I (* I denotes an identifier *)
IExp: IE: $:= z \mid I \mid (IE_1 + IE_2) \mid \ldots$
BExp: BE: $:= (IE_1 < IE_2) \mid \ldots$
Decl: Δ : $:= \Delta_c \Delta_v \Delta_p$
Δ_c: $:= \varepsilon \mid$ **const** $I_1 = z_1, \ldots, I_n = z_n$;
Δ_v: $:= \varepsilon \mid$ **var** $I_1, \ldots, I_n$;
Δ_p: $:= \varepsilon \mid$ **proc** I_1; B_1; ...; I_n; B_n;
Cmd: Γ: $:= I := IE \mid$ **begin** Γ_1; ...; Γ_n **end** $\mid$ **if** BE **then** $\Gamma \mid$ **while** BE **do** $\Gamma \mid$ **call** I
Block: B: $:= \Delta\Gamma$
Prog: P: $:=$ **in/out** $I_1, \ldots, I_n$; B.

where n always is a positive integer: $n \geqslant 1$.

We could easily extend the language without affecting the translation technique as demonstrated during a compiler construction course.

Example program PF

```
in/out X;
   var E;
   proc F;
      if (1 < X) then
         begin E := (E * X);
          X := (X - 1); call F
         end;
   begin E := 1; call F;
          X := E
   end.
```

B_F $= \Gamma_F$ } Δ_p } Δ ; } Γ

Note that we used subtraction and multiplication without explicit syntactic specification.

1.2. Denotational semantics of PL/O. The formal definition will satisfy the following conditions for identifiers:

– Identifiers defined by a declaration Δ are pairwise distinct.

– An identifier occurring in the command part Γ of a block $\Delta\Gamma$ has to be declared in Δ or in the declaration part of an outer block.

In particular, one may declare mutually recursive procedures.

– It is possible to redeclare identifiers where the innermost level will be the defining one.

– Static scope: The semantics of a procedure call is defined with respect to the environment of the corresponding procedure declaration, not of the procedure call.

Semantic domains

$\mathbf{Z} := \{z \mid z \text{ is an integer}\}$.
$\mathbf{B} := \{\mathbf{true}, \mathbf{false}\}$,
$\mathbf{Loc} := \{\alpha_i \mid i \geqslant 1\}$ is an infinite set of *locations*.
$S := \{\sigma \mid \sigma\colon \mathbf{Loc} \to \mathbf{Z}\}$ is the set of *states*.
$C := \{\theta \mid \theta\colon S \to S\}$ is the set of *state transformations*. They will be used as procedure values.
$U := \{\varrho \mid \varrho\colon \mathbf{Ide} \to (\mathbf{Z} \cup \mathbf{Loc} \cup C)\}$ is the set of environments.

We have chosen sets and partial functions instead of cpo's and continuous functions since this suffices for our purposes. So we may have e.g. the empty state $\sigma_\emptyset$ and the empty environment $\varrho_\emptyset$.

Auxiliary functions

To update a function $f\colon A \to B$ for different arguments $a_1, \ldots, a_n \in A$ with new values $b_1, \ldots, b_n \in B$ we write:

$f[a_1/b_1, \ldots, a_n/b_n]\colon A \to B$
$a \mapsto \mathbf{if}\ a = a_i\ \mathbf{then}\ b_i\ \mathbf{else}\ f(a)$

dis: $\mathbf{Decl} \to \mathbf{B}$ tests distinctness of identifiers:
dis $(\Delta) = \mathbf{true}$ iff all identifiers declared in Δ are pairwise different.
last: $S \to N$ determines the last location used in a given state σ:
last $(\sigma) = n$ iff $\sigma(\alpha_n) \in \mathbf{Z}$
and $\sigma(\alpha_{n+k})$ is undefined for all $k > 0$
and **last** $(\sigma_\emptyset) := 0$.

Note that in this paper a state σ will always be of finite support. Hence, **last** (σ) will always be defined.

Semantic functions

Integer expressions denote integers provided that appropriate environments and states are given.

$$\boxed{\mathbf{E}_i\colon \mathbf{IExp} \times U \times S \to \mathbf{Z}}$$

$$\mathbf{E}_i[\![z]\!]\varrho\sigma := z$$
$$\mathbf{E}_i[\![I]\!]\varrho\sigma := \mathbf{if}\ \varrho(I) = z\ \mathbf{then}\ z$$
$$\mathbf{if}\ \varrho(I) = \alpha\ \mathbf{then}\ \sigma(\alpha).$$

For economy of notation we assume that a missing **else**-clause means that the function is undefined in that remaining case.

$$\mathbf{E}_i[\![(IE_1 + IE_2)]\!]\varrho\sigma := \mathbf{E}_i[\![IE_1]\!]\varrho\sigma + \mathbf{E}_i[\![IE_2]\!]\varrho\sigma.$$

Similarly, we get the semantics of boolean expressions.

$$\boxed{\mathbf{E}_b\colon \mathbf{BExpr} \times U \times S \to \mathbf{B}}$$

$$\mathbf{E}_b[\![(IE_1 < IE_2)]\!]\varrho\sigma := \textbf{if } \mathbf{E}_i[\![IE_1]\!]\varrho\sigma < \mathbf{E}_i[\![IE_2]\!]\varrho\sigma \textbf{ then true}$$
$$\textbf{if } \mathbf{E}_i[\![IE_1]\!]\varrho\sigma \geqslant \mathbf{E}_i[\![IE_2]\!]\varrho\sigma \textbf{ then false}$$

A declaration may change its environment and in case of a variable declaration also the state.

$$\boxed{\mathbf{D}\colon \mathbf{Decl} \times U \times S \to U \times S}$$

$$\mathbf{D}[\![\Delta_c\,\Delta_v\,\Delta_p]\!]\varrho\sigma := \textbf{if dis } (\Delta_c\,\Delta_v\,\Delta_p) = \textbf{true}$$
$$\textbf{then } \mathbf{D}[\![\Delta_p]\!](\mathbf{D}[\![\Delta_v]\!](\mathbf{D}[\![\Delta_c]\!]\varrho\sigma))$$
$$\mathbf{D}[\![\varepsilon]\!]\varrho\sigma := \varrho\sigma$$
$$\mathbf{D}[\![\textbf{const } I_1 = z_1, \ldots, I_{n+1} = z_{n+1};\,]\!]\varrho\sigma :=$$
$$\textbf{if } \mathbf{D}[\![\textbf{const } I_1 = z_1, \ldots, I_n = z_n;\,]\!]\varrho\sigma = \varrho'\sigma'$$
$$\textbf{then } \varrho'[I_{n+1}/z_{n+1}]\sigma'$$
$$\mathbf{D}[\![\textbf{var } I_1, \ldots, I_{n+1};]\!]\varrho\sigma := \textbf{if } \mathbf{D}[\![\textbf{var } I_1, \ldots, I_n]\!]\varrho\sigma = \varrho'\sigma'$$
$$\textbf{and last } (\sigma') = l$$
$$\textbf{then } \varrho'[I_{n+1}/\alpha_{l+1}]\sigma'[\alpha_{l+1}/0]$$
$$\mathbf{D}[\![\textbf{proc } I_1; B_1; \ldots; I_n; B_n;\,]\!]\varrho\sigma := \varrho[I_1/\theta_1, \ldots, I_n/\theta_n]\sigma$$
$$\textbf{where } \theta_i = \mathbf{B}[\![B_i]\!]\varrho[I_1/\theta_1, \ldots, I_n/\theta_n] \quad (1 \leqslant i \leqslant n)$$
with least fixed-point semantics.

For the semantics of a procedure declaration we assume static scope, i.e., the declaration environment determines the procedure meaning.

A command denotes a state transformation that depends on an environment.

$$\boxed{\mathbf{C}\colon \mathbf{Cmd} \times U \times S \to S}$$

$$\mathbf{C}[\![I := IE]\!]\varrho\sigma := \textbf{if } \varrho(I) = \alpha \textbf{ then } \sigma[\alpha/\mathbf{E}_i[\![IE]\!]\varrho\sigma]$$
$$\mathbf{C}[\![\textbf{begin } \Gamma_1; \ldots; \Gamma_n \textbf{ end}]\!]\varrho\sigma := \mathbf{C}[\![\Gamma_n]\!]\varrho(\ldots(\mathbf{C}[\![\Gamma_1]\!]\varrho\sigma)\ldots)$$
$$\mathbf{C}[\![\textbf{if } BE \textbf{ then } \Gamma]\!]\varrho\sigma := \textbf{if } \mathbf{E}_b[\![BE]\!]\varrho\sigma = \textbf{true then } \mathbf{C}[\![\Gamma]\!]\varrho\sigma$$
$$\textbf{if } \mathbf{E}_b[\![BE]\!]\varrho\sigma = \textbf{false then } \sigma$$
$$\mathbf{C}[\![\underbrace{\textbf{while } BE \textbf{ do } \Gamma}_{\Gamma'}]\!]\varrho\sigma := \textbf{if } \mathbf{E}_b[\![BE]\!]\varrho\sigma = \textbf{true then } \mathbf{C}[\![\Gamma']\!]\varrho\,(\mathbf{C}[\![\Gamma]\!]\varrho\sigma)$$
$$\textbf{if } \mathbf{E}_b[\![BE]\!]\varrho\sigma = \textbf{false then } \sigma$$
with least fixed-point semantics
$$\mathbf{C}[\![\textbf{call } I]\!]\varrho\sigma := \textbf{if } \varrho(I) = \theta \textbf{ then } \theta(\sigma).$$

Here, one realizes the advantage of mathematical semantics: the meaning of a procedure call is fully determined by a symbol table look-up because in the symbol table the procedure identifier I was bound to the corresponding state transformation during declaration as required by our static scope assumption.

A block also denotes a state transformation. However, the corresponding environment will be updated by the local declaration part.

$$\boxed{\mathbf{B} = \mathbf{Block} \times U \times S \to S}$$

$$\mathbf{B}[\![\Delta\Gamma]\!]\varrho\sigma := \mathbf{C}[\![\Gamma]\!](\mathbf{D}[\![\Delta]\!]\varrho\sigma)$$

Finally, the meaning of a program is understood as a transformation of integer sequences mapping input values onto output values.

$$\mathbf{M}: \text{Prog} \times Z^* \to Z^*$$

$\mathbf{M}[\![\textbf{in/out}\ I_1, \ldots, I_n; B.]\!](z_1, \ldots, z_m) :=$
if $m = n$ **then** $(\sigma(\alpha_1), \ldots, \sigma(\alpha_n))$
where $\sigma = \mathbf{B}[\![B]\!]\varrho_\emptyset[I_1/\alpha_1, \ldots, I_n/\alpha_n]\sigma_\emptyset[\alpha_1/z_1, \ldots, \alpha_n/z_n]$.

It should be noted that in fact these semantic functions are defined by structural induction (or recursion) since function calls on right-hand sides of equations have arguments which are syntactic parts of the left-hand side argument. In case of the **while**-command this property could be obtained by explicitly using the fixed-point operator.

2. The target language SC

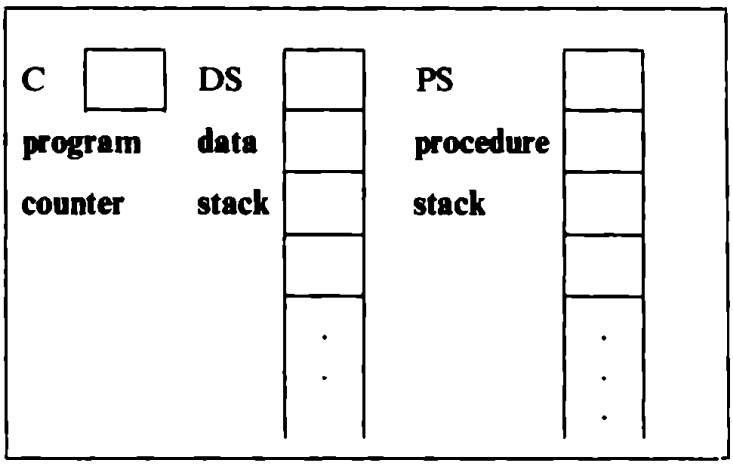
Since we describe a compiler front-end for a block-structured language we choose a stack machine code SC as target language.

Stacks had been invented by Bauer and Samelson [7] for proper expression evaluation. Dijkstra [3] generalized this technique for compiling blocks and procedures, including recursive procedures. This problem arose during the implementation of ALGOL 60. For its solution Dijkstra proposed a stack where global variables could be handled by pointers (static links) which allow to read inner stack elements.

Here, we shall use a stack code SC which is only a slight modification from [9].

The store of the underlying stack machine has three components: a program counter C, a data stack DS, and a procedure stack PS.

stack machine



We prefer to keep data and procedure stack separate leaving their efficient organization on a real machine as a back-end task.

2.1. Syntax of SC. The set **Instr** of SC-instructions contains control instructions, data stack instructions, and procedure stack instructions.

a) control instructions:

JMP(a) (* jump to a *)
JMC(a) (* jump on condition to a *)

for each $a = n_1 \cdot \ldots \cdot n_r \in N^*$. The use of tree-structured jump addresses simplifies their inductive computation during translation;

b) data stack instructions:

ADD (* add *)
LIT(z) (* load immediately z *)
LT (* less than *)

for each $z \in Z$;

c) procedure stack instructions:

CREATE (l, a, t) (* create *)
RET (* return *)
LOD(l, o) (* load *)
STO(l, o) (* store *)

for each $l, o, t \in N$ and $a \in N^*$.

The set **Code** of SC-programs consists of instruction sequences where in addition each instruction may be labelled by several jump addresses. This more general labelling supports again our inductive translation technique:

$$M := w_1 : \mathrm{instr}_1 ; \ldots ; w_n : \mathrm{instr}_n \in \mathbf{Code}$$

if $n \geqslant 1$, $\mathbf{instr}_1, \ldots, \mathbf{instr}_n$ are SC-instructions and $w_1, \ldots, w_n \in (N^*)^*$ are address sequences where an address must not occur twice, i.e.: from $w_1 \cdot w_2 \cdot \ldots \cdot w_n = a_1 : a_2 : \ldots : a_m$ with $a_i \in N^*$ and $j \neq k$ it follows that $a_j \neq a_k$.

In order to facilitate the formal semantics of SC we assume that the stack machine on loading the machine program replaces all jump addresses by linear standard addresses. Thereby, an SC-program $M = w_1 : \mathbf{instr}_1 ; \ldots ; w_n : \mathbf{instr}_n$ will be transformed into standard address form:

$$M' = 1 : \mathbf{instr}'_1 ; \ldots ; n : \mathbf{instr}'_n$$

where $\mathbf{instr}'_i$ results from $\mathbf{instr}_i$ by proper replacement of jump addresses. More precisely, if an address occurs in the left-hand address sequence w_i, it is replaced by i, otherwise, it becomes 0, the standard stop address.

EXAMPLE. The following SC-program will turn out as the translation of our example program PF $\in$ **Prog**. Here, we assume to have two additional data stack instructions:

and	SUB	for subtraction,
	MULT	for multiplication.

For better readability we put addresses into brackets.

an SC-*program*	and its *standard address form*
CREATE (0, 0, 1);	[1]: CREATE (0, 0, 1);
JMP (1);	[2]: JMP (18);
[1.1.1]: LIT (1);	[3]: LIT (1);
LOD (2, 1);	[4]: LOD (2, 1);
LT;	[5]: LT
JMC (1.1.1.2);	[6]: JMC (17);
LOD (1, 1);	[7]: LOD (1, 1);
LOD (2, 1);	[8]: LOD (2, 1);
MULT;	[9]: MULT;
STO (1, 1);	[10]: STO (1, 1);
LOD (2, 1)	[11]: LOD (2, 1);
LIT (1);	[12]: LIT (1);
SUB;	[13]: SUB;
STO (2, 1);	[14]: STO (2, 1);
CREATE (1, 1.1.1.2.1.3, 0);	[15]: CREATE (1, 17, 0);
JMP (1.1.1);	[16]: JMP (3);
[1.1.1.2.1.3, 1.1.1.2]:	
RET;	[17]: RET;
[1]: LIT (1);	[18]: LIT (1);
STO (0, 1);	[19]: STO (0, 1);
CREATE (0, 1.2, 0);	[20]: CREATE (0, 22, 0);
JMP (1.1.1);	[21]: JMP (3);
[1.2]: LOD (0, 1);	[22] LOD (0, 1);
STO (1, 1);	[23] STO (1, 1);
RET	[24] RET

2.2. Semantics of SC. The semantics of an SC-program $M \in$ **Code** will be defined with respect to its standard address form M'. Therefore, it suffices to consider natural numbers as jump addresses, and we define the set STORE of stack machine states as follows:

$\text{STORE} := \text{C} \times \text{DS} \times \text{PS}$

where $\text{C} := N$

$\text{DS} := Z^*$

$\text{PS} := Z^*$

A state $s \in$ STORE will be denoted by

$$s = (c, d, p)$$

with $d = d \cdot 1 : d \cdot 2 : \ldots : d \cdot r$

$p = p \cdot 1 : p \cdot 2 : \ldots : p \cdot t$

This notation indicates that for data and procedure stacks the top is handled on the left.

Before we turn to the formal semantics of instructions and programs, we introduce an auxiliary function **base** for indirect address calculation on the procedure stack by means of static links.

base: PS $\times N \rightarrow N$,

base $(p, o) := 1$,

base $(p, l+1) :=$ **base** $(p, l) + p \cdot$ **base** (p, l).

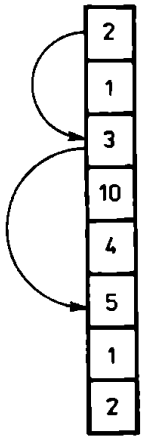
EXAMPLE. For $p = 2{:}1{:}3{:}10{:}4{:}5{:}1{:}2$ we have

base $(p, 0) = 1$,

base $(p, 1) =$ **base** $(p, 0) + p \cdot$ **base** $(p, o) = 1 + p \cdot 1 = 3$,

base $(p, 2) =$ **base** $(p, 1) + p \cdot$ **base** $(p, 1) = 3 + p \cdot 3 = 6$.

This means that certain procedure stack entries are interpreted as relative addresses (static links), e.g., $p \cdot 3 = 3$ points to $p \cdot 6$.



Instruction semantics

Each instruction **instr** $\in$ **Instr** denotes a state transformation

$$\mathbf{I}[\![\textbf{instr}]\!]: \text{STORE} \rightarrow \text{STORE};$$

a) control instructions:

$\mathbf{I}[\![\text{JMP}(n)]\!](c, d, p) := (n, d, p)$,

$\mathbf{I}[\![\text{JMC}(n)]\!](c, d, p) :=$ **if** $d = 0 : d'$ **then** (n, d', p),

if $d = 1 : d'$ **then** $(c+1, d', p)$.

Note that the conditional jump instruction requires for execuion a boolean value on the data stack where 0 means false and 1 true:

b) data stack instructions:

$$\begin{aligned}
\mathbf{I}[\![\mathrm{ADD}]\!](c, d, p) &:= \textbf{if } d = z_1\colon z_2\colon d' \\
&\quad \textbf{then } (c+1, z_1+z_2\colon d', p), \\
\mathbf{I}[\![\mathrm{LIT}(z)]\!](c, d, p) &:= (c+1, z\colon d, p), \\
\mathbf{I}[\![\mathrm{LT}]\!](c, d, p) &:= \textbf{if } d = z_2\colon z_1\colon d' \\
&\quad \textbf{then } (c+1, b\colon d', p) \\
&\quad \textbf{where } b := \begin{cases} 0 \textbf{ if } z_1 \geqslant z_2, \\ 1 \textbf{ if } z_1 < z_2; \end{cases}
\end{aligned}$$

c) procedure stack instructions:

$$\mathbf{I}[\![\mathrm{CREATE}\ (l, a, t)]\!](c, d, p) :=$$

$$(c+1, d, \textbf{base}(p, l)+t+2\colon t+2\colon a\colon \overbrace{0\colon \ldots \colon 0}^{t}\colon p).$$

Here, a new activation record is created on top of the procedure stack. **base** $(p, l)+t+2$ is the so-called static link which points to the beginning of the environment of that record, i.e., to the next active record. The dynamic link $t+2$ points to the next record. a is the return address, and t is the number of local storage locations. They are initialized by 0.

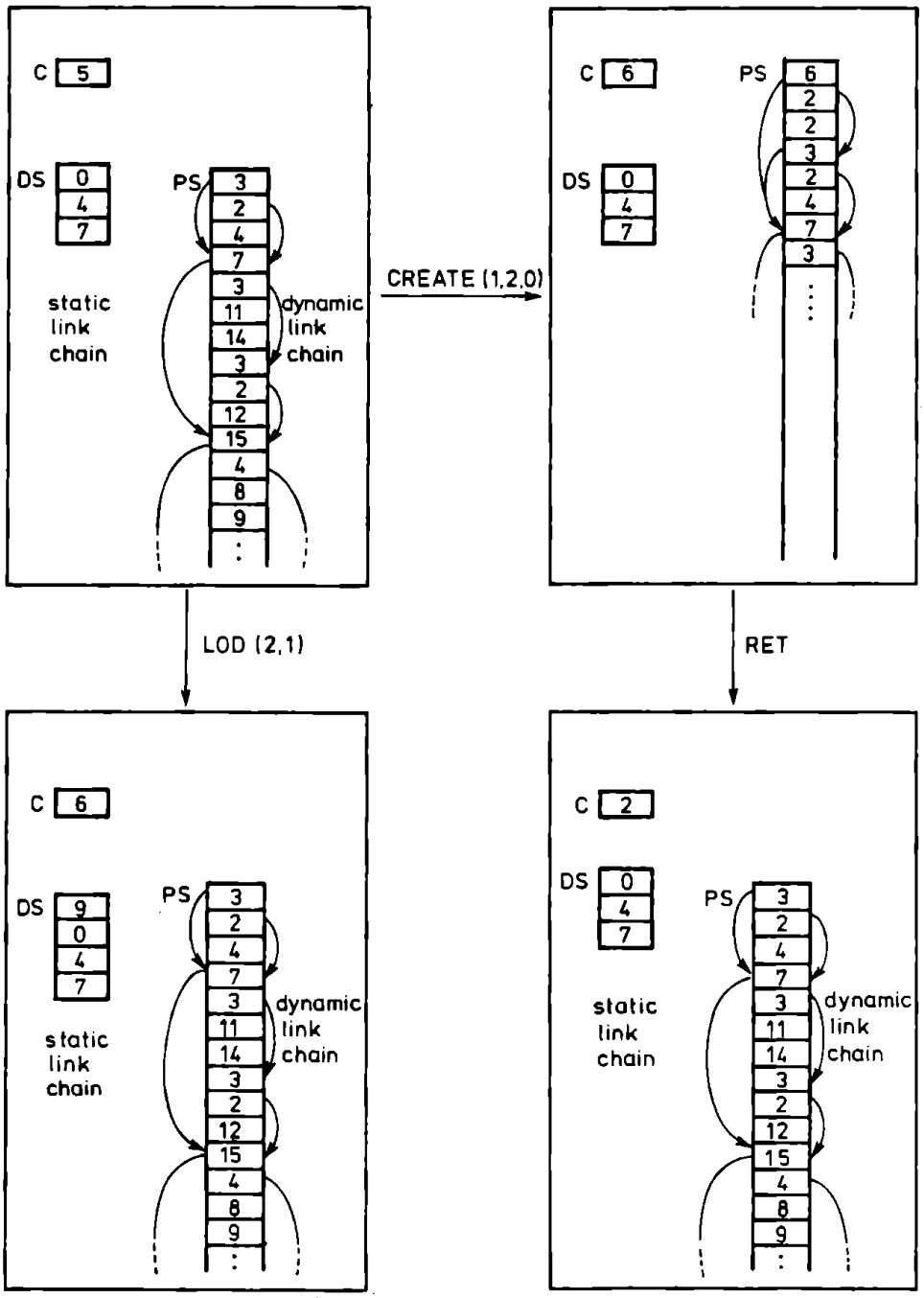
$$\begin{aligned}
\mathbf{I}[\![\mathrm{RET}]\!](c, d, p) &:= \textbf{if } p = sl\colon dl\colon a\colon z_1\colon \ldots \colon z_{dl-2}\colon p' \\
&\quad \textbf{then } (a, d, p').
\end{aligned}$$

Execution of a return instruction requires an appropriate activation record on the procedure stack. The dynamic link dl determines the length of that record which is $dl+1$. The counter is set to the return address a and the record is popped.

Load and store instructions cause data transfer between data and procedure stack. Level l and offset o are parameters describing a location on the procedure stack.

$$\begin{aligned}
\mathbf{I}[\![\mathrm{LOD}(l, o)]\!](c, d, p) &:= (c+1, z\colon d, p) \\
&\quad \textbf{where } z = p\cdot(\textbf{base } (p, l)+o+2), \\
\mathbf{I}[\![\mathrm{STO}(l, o)]\!](c, d, p) &:= \textbf{if } d = z\colon d' \textbf{ then } (c+1, d', p') \\
&\quad \textbf{where } p'\cdot(\textbf{base}(p, l)+o+2) = z \\
&\quad \textbf{and } p'\cdot i = p\cdot i \textbf{ otherwise}.
\end{aligned}$$

We illustrate the effect of these procedure stack·instructions by the following example:



Semantics of SC-programs

Let $M \in$ **Code** be an SC-program in standard address form, i.e.,

$$M = 1\text{: } \mathbf{instr}_1; \ldots; n\text{: } \mathbf{instr}_n$$

and all jump labels are in $\{0, 1, 2, \ldots, n\}$.

First, we associate with M its *transition semantics*:

$$\mathbf{T}[\![M]\!]\text{: STORE} \rightarrow \text{STORE},$$
$$\mathbf{T}[\![M]\!](c, d, p) := \mathbf{if}\ 1 \leqslant c \leqslant n \ \mathbf{then}\ \mathbf{I}[\![\mathbf{instr}_c]\!](c, d, p).$$

Next, we generate its *continuation semantics* by iteration:

$$\mathbf{C}[\![M]\!]\colon \mathrm{STORE} \to \mathrm{STORE},$$
$$\mathbf{C}[\![M]\!](c, d, p) := \textbf{if } c = 0 \textbf{ then } (c, d, p) \textbf{ else } \mathbf{C}[\![M]\!](\mathbf{T}[\![M]\!](c, d, p)).$$

Finally, the *meaning* $\mathbf{M}[\![M]\!]$ will be a transformation of integer sequences:

$$\mathbf{M}[\![M]\!]\colon Z^* \to Z^*,$$
$$\mathbf{M}[\![M]\!] := \textbf{output} \circ \mathbf{C}[\![M]\!] \circ \textbf{input},$$
where $\textbf{input}\colon Z^* \to \mathrm{STORE},$
$$\textbf{input}\ (z_1, \ldots, z_m) := (1, \varepsilon, 0\colon 0\colon 0\colon z_1\colon \ldots\colon z_m)$$

sets control to 1 calling for the first instruction and generates an activation record for given input values,

and $\textbf{output}\colon \mathrm{STORE} \to Z^*,$
$$\textbf{output}\ (o, \varepsilon, 0\colon 0\colon 0\colon z_1\colon \ldots\colon z_m) := (z_1, \ldots, z_m)$$

reads output values from the last activation record.

Note that the data stack will only be used for intermediate computations and further that computations may only stop with address 0.

It remains to generalize this meaning to arbitrary SC-programs. So, if $M \in \mathbf{Code}$, we simply define

$$\mathbf{M}[\![M]\!] := \mathbf{M}[\![M']\!]$$

where M' is the standard address form of M.

3. The translation of PL/O-programs into SC-programs

We are now well prepared for defining the translation of source programs into target programs. As with the semantics of PL/O-programs we shall proceed by structural induction. However, in denotational semantics we are allowed to translate our programs into a comfortable mathematical world of meanings where in particular we may use infinite objects. For instance, a procedure identifier is bound to its full state transformation in the environment.

Now, we have to translate into intermediate stack code. Instead of infinite environments we are forced to use symbol tables where only a finite amount of information may be stored for a procedure identifier. Here, a central task will be the proper organization of global storage for a procedure call by means of static links.

By structural recursion on the first argument we simultaneously define the following translation mappings:

$$\begin{aligned}
\textbf{trans}&: \textbf{Prog} \to \textbf{Code}\\
\textbf{blocktrans}&: \textbf{Block} \times \textbf{Tab} \times \textbf{Adr} \times \textbf{Lev} \to \textbf{Code}\\
\textbf{decltrans}&: \textbf{Decl} \times \textbf{Tab} \times \textbf{Adr} \times \textbf{Lev} \to \textbf{Code}\\
\textbf{cmdtrans}&: \textbf{Cmd} \times \textbf{Tab} \times \textbf{Adr} \times \textbf{Lev} \to \textbf{Code}\\
\textbf{iexptrans}&: \textbf{IExpr} \times \textbf{Tab} \times \textbf{Lev} \to \textbf{Code}\\
\textbf{bexptrans}&: \textbf{BExpr} \times \textbf{Tab} \times \textbf{Lev} \to \textbf{Code}
\end{aligned}$$

These functions are partial since the syntactic argument may not fulfil semantic constraints, e.g. identifier requirements.

As parameters we use

$$\begin{aligned}
l \in \textbf{Lev} &:= N \quad \text{for the level of nested blocks,}\\
a \in \textbf{Adr} &:= N^* \quad \text{for an unused jump label, and}\\
st \in \textbf{Tab} &\quad\quad \text{for a symbol table}
\end{aligned}$$

where

$$\begin{aligned}
st: \textbf{Ide} \to\ & (\{\textbf{const}\} \times Z)\\
& \cup (\{\textbf{var}\} \times \textbf{Lev} \times \textbf{Offset})\\
& \cup (\{\textbf{proc}\} \times \textbf{Lev} \times \textbf{Adr} \times \textbf{Size})
\end{aligned}$$

where Offset $:= N$ **and Size** $:= N$.

A symbol table will always have a finite number of entries only, i.e., $|\textbf{def}(st)| < \infty$. With a constant identifier we associate the corresponding integer, with a variable identifier its declaration level $dl \in \textbf{Lev}$ and its relative declaration position $o \in \textbf{Offset}$, and with a procedure identifier its declaration level $dl \in \textbf{Lev}$, an address $a \in \textbf{Adr}$ indicating the first instruction of the procedure code, and the number $s \in \textbf{Size}$ of local variables.

Auxiliary functions

The function

$$\textbf{size}: \textbf{Block} \to N$$

defined by

$$\begin{aligned}
&\textbf{size}\ (\Delta_c\, \Delta_p\, \Gamma) := 0,\\
&\textbf{size}\ (\Delta_c\, \textbf{var}\, I_1, \ldots, I_n; \Delta_p\, \Gamma) := n
\end{aligned}$$

will be used to compute the local storage need of a procedure.

The function

$$\textbf{update}: \textbf{Decl} \times \textbf{Tab} \times \textbf{Adr} \times \textbf{Lev} \to \textbf{Tab}$$

performs symbol table entries for a declaration Δ with respect to an unused jump label $a \in \textbf{Adr}$ and the block nesting level $l \in \textbf{Lev}$ of Δ.

update $(\Delta_c \Delta_v \Delta_p, st, a, l) :=$ **if dis** $(\Delta_c \Delta_v \Delta_p) =$ **true then**
update $(\Delta_p,$ **update** $(\Delta_v,$ **update** $(\Delta_c, st, a, l), a, l), a, l)$,

update $(\varepsilon, st, a, l) := st$,

update (**const** $I_1 = z_1, \ldots, I_n = z_n$; , $st, a, l) :=$
$st\,[I_1/(\textbf{const}, z_1), \ldots, I_n/(\textbf{const}, z_n)]$,

update (**var** $I_1, \ldots, I_n$; , $st, a, l) :=$
$st\,[I_1/(\textbf{var}, l, 1), \ldots, I_n/(\textbf{var}, l, n)]$,

update (**proc** $I_1; B_1; \ldots; I_n; B_n$; , $st, a, l) :=$
$st\,[I_1/(\textbf{proc}, l, a \cdot 1,$ **size** $(B_1))$.
$\vdots$
$I_n/(\textbf{proc}, l, a \cdot n,$ **size** $(B_n))]$.

Here we see that a constant identifier is bound to its value, a variable identifier to its declaration level and offset (relative position in the variable declaration list), and a procedure identifier to its declaration level, to a new address and the number of local variables. We shall see in a moment that this new address will in fact be the start address for the corresponding block code.

Initial symbol table

As we defined the input mapping for our stack machine by

$$\textbf{input}: Z^* \to \text{STORE}$$
$$\text{with } \textbf{input}\ (z_1, \ldots, z_m) := (1, \varepsilon, 0{:}\ 0{:}\ 0{:}\ z_1{:}\ \ldots{:}\ z_m)$$

we shall use for the variables $I_1, \ldots, I_n$ of the input/output-declaration **in/out** $I_1, \ldots, I_n$; the following *initial symbol table*

$$st\ \langle I_1, \ldots, I_n \rangle \in \textbf{Tab}$$
$$\text{with } st\ \langle I_1, \ldots, I_n \rangle (I_j) := (\text{var}, 0, j).$$

This is because the input/output-declaration is assumed to have level 0.

Now, we can construct the translation mappings by structural recursion on the syntactic parts of source programs.

trans $(in/out\ I_1, \ldots, I_n; B.) :=$ CREATE $(0, 0,$ **size**$(B))$;
JMP(1);
blocktrans $(B, st\ \langle I_1, \ldots, I_n \rangle, 1, 1)$

The first instruction creates a new activation record for execution of block B. This activation record is linked statically and dynamically to the input/output-activation record. The standard stop address 0 will be used when the last

instruction of the block translation, namely a return instruction, is executed: control will be set to 0 and the activation record will be removed from the stack.

The jump instruction sets control to 1 which is the address of the first instruction of the command translation. This is guaranteed by calling the function *blocktrans* with address parameter 1. What follows is the translation of block B with respect to the initial symbol table, address parameter 1, and block level l.

blocktrans $(\Delta\Gamma, st, a, l) :=$
  decltrans $(\Delta,$ **update** $(\Delta, st, a\cdot 1, l), a\cdot 1, l),$
$[a]$: **cmdtrans** $(\Gamma,$ **update** $(\Delta, st, a\cdot 1, l), a\cdot 2, l)$
  RET.

At the beginning, control enters the command translation. Here, a procedure call yields a jump to some procedure code produced by the declaration translation. Note that tree structured addresses allow a simple generation of new addresses $a\cdot 1$ and $a\cdot 2$ from a,

decltrans $(\Delta_c \Delta_v \Delta_p, st, a, l) :=$
  decltrans $(\Delta_p, st, a, l),$
decltrans $(\varepsilon, st, a, l) := \varepsilon,$
decltrans (**proc** $I_1; B_1; \ldots; I_n; B_n;$, $st, a, l) :=$
  blocktrans $(B_1, st, a\cdot 1, l+1);$
  $\vdots$
  blocktrans $(B_n, st, a\cdot n, l+1).$

So, a procedure declaration causes the generation of code. Note that now the block level increases by one. Moreover, since **decltrans** and **update** were called with the same address parameter (see the definition of blocktrans), it is guaranteed that the symbol table generated by **update** contains correct procedure code addresses;

cmdtrans $(I := IE, st, a, l) :=$ **if** $st(I) =$ (**var**, $dl, o)$ **then**
  iexptrans (IE, st, l)
  STO $(l-dl, o).$

Integer expression evaluation produces an expression value on top of the data stack. This value is stored in the procedure stack. The correct activation record is determined by the block level difference between occurrence and declaration of variable I,

cmdtrans (**begin** $\Gamma_1; \ldots; \Gamma_n$ **end** , $st, a, l) :=$
  cmdtrans $(\Gamma_1, st, a\cdot 1, l)$
  $\vdots$
  cmdtrans $(\Gamma_n, st, a\cdot n, l)$

```
cmdtrans (if BE then Γ, st, a, l) :=
        bexptrans (BE, st, l)
        JMC(a);
        cmdtrans (Γ, st, a·1, l)
        [a]:
```

After evaluating BE on the data stack, a conditional jump instruction decides whether the code for Γ will be executed or not.

```
cmdtrans (while BE do Γ, st, a, l) :=
        [a]: bexptrans (BE, st, l)
             JMC (a·1);
             cmdtrans (Γ, st, a·2, l)
             JMP(a);
        [a·1]:
```

Note that because of inductive address generation an instruction nay get various addresses.

```
cmdtrans (call I, st, a, l) := if st(I) = (proc, dl, ca, n) then
        CREATE (l−dl, a, n);
        JMP(ca);
        [a]:
```

A procedure call is translated as follows: first, a new activation record is created. The level difference $l-dl$ explains where its environment on the procedure stack begins, i.e., where global variable locations are found. The label a works as return address, and the jump instruction sets control to the initial procedure code address *ca* found in the symbol table.

The remaining translation of expressions is straightforward.

```
iexptrans (z, st, l) := LIT(z);
iexptrans (I, st, l) := if st(I) = (const, z) then
                        LIT(z);
                        if st(I) = (var, dl, o) then
                        LOD(l−dl, o);

iexptrans ((IE1 + IE2), st, l) := iexptrans (IE1, st, l)
                                  iexptrans (IE2, st, l)
                                  ADD;
bexptrans ((IE1 < IE2), st, l) := iexptrans (IE1, st, l)
                                  iexptrans (IE2, st, l)
                                  LT.
```

This completes our functional compiler description. Of course, it remains to prove that this translation is correct in the following sense:

THEOREM. *For all* $P = \textbf{in/out}\ I_1, \ldots, I_n$; $B. \in \textbf{Prog}$ *and for all* $z = (z_1, \ldots, z_n) \in \mathbf{Z}^n$ *we have*

$$\mathbf{M}[\![P]\!](z) = \mathbf{M}[\![\textbf{trans}(P)]\!](z).$$

EXAMPLE. Instead of giving a formal proof here we apply the translation mapping to our example program PF:

trans (**in/out** X; $\Delta\Gamma$) = CREATE (0, 0, 1);
JMP (1);
blocktrans ($\Delta\Gamma$, $st\langle X\rangle$, 1, 1)
where $st\langle X\rangle(X)$ = (**var**, 0, 1)

blocktrans($\Delta\Gamma$, $st\langle X\rangle$, 1, 1) = **decltrans** (Δ, **update**(Δ, $st\langle X\rangle$, 1.1, 1), 1.1, 1)
[1]: **cmdtrans** (Γ, **update**(Δ, $st\langle X\rangle$, 1.1, 1). 1.2, 1)
RET

update (Δ, $st\langle X\rangle$, 1.1, 1) = $st\langle X\rangle[E/(\textbf{var}, 1, 1), F/(\textbf{proc}, 1, 1.1.1, 0)]$
$=: st1$

decltrans (Δ, $st1$, 1.1, 1) = **blocktrans** (B_F, $st1$, 1.1.1, 2);
= [1.1.1]: **cmdtrans** (Γ_F, $st1$, 1.1.1.2, 2)
RET;

cmdtrans (Γ_F, $st1$, 1.1.1.2, 2) = **bexptrans** ((1 < X), $st1$, 2)
JMC (1.1.1.2);
cmdtrans (**begin**...**end**, $st1$, 1.1.1.2.1, 2)
[1.1.1.2]:

bexptrans ((1 < X), $st1$, 2) = LIT(1);
LOD(2, 1);
LT;

cmdtrans(**begin**...**end**, $st1$, 1.1.1.2.1, 2) = **cmdtrans** ($E := (E * X)$, $st1$, 1.1.1.2.1.1, 2)
cmdtrans ($X := (X-1)$, $st1$, 1.1.1.2.1.2, 2)
cmdtrans (**call** F, $st1$, 1.1.1.2.1.3, 2)
= LOD(1, 1);
LOD(2, 1);
MULT;
STO(1, 1);
LOD(2, 1);
LIT(1);
SUB;
STO(2, 1);
CREATE(1, 1.1.1.2.1.3, 0);
JMP(1.1.1);
[1.1.1.2.1.3]:

cmdtrans $(\Gamma, st1,1.2, 1)$ = LIT(1);
STO(0, 1);
CREATE(0, 1.2, 0);
JMP(1.1.1);
[1.2]: LOD(0, 1);
STO(1, 1).

From this evaluation of function calls one sees that the resulting SC-program is in fact the example program of Section 2.1.

Conclusion

Our case study shows that for defining the translation mapping of a compiler front-end structural-recursive functions with parameters offer greater flexibility than just homomorphisms. In [8] Thatcher, Wagner and Wright demonstrate that for a source language with statements and expressions the homomorphic technique in fact suffices for defining semantics and translation. Moreover, they obtain an algebraic compiler correctness proof based on initiality. We believe that for block-structured languages with recursive procedure declarations this goal cannot be achieved by homomorphisms since the meaning of a statement now depends on the environment given by a declaration. For the same reason a description by attribute grammars would require inherited attributes. Now, Chirica and Martin show in [1] that inherited attributes can be eliminated, but at the price of enlarging the semantic algebra by means of function domains. The idea is certainly applicable to denotational semantics yielding algebraic semantics. However, the translation mapping has a *fixed* target domain. We therefore believe that structural-recursive functions with parameters form an appropriate and natural class of functions for describing more complex language translations.

The same type of functions has already been used for giving denotational semantics. However, finiteness constraints of compiling demand for a more subtle treatment. Despite this increase of technical complexity it should be clear that our method supports an inductive compiler correctness proof. We hope to develop such a proof in a forthcoming paper and compare it with the different approach of Jones and Lucas [4].

Succinctness and modularity are further advantages, especially for teaching compiler construction. Generation of intermediate code can be treated with the same formal rigor as lexical and syntactic analysis. In a course on compiler construction we have demonstrated that other language constructs such as procedures with parameters or data structures can be handled in the same way.

Finally, it should be noted that at least a rapid compiler prototype is

obtainable using any programming language which accepts recursive function definitions over tree-structured domains as programs. E.g., LISP, PROLOG and ML are appropriate candidates for this task. However, since their compilers will not exploit the particular recursion structure of these translation functions, the resulting implementation might be less efficient than a more direct, iterative compiler description.

This observation suggests to develop a functional language based on structural recursion. A first step in this direction is Klaeren's language SRDL [5]. The close relationship between this class of recursive functions and attribute grammars as observed by Courcelles and Franchi-Zannettacci [2] indicates a possible approach to more efficient implementations by carrying over suitable attribute evaluation techniques. We hope to develop these ideas in further research.

Acknowledgements. I would like to thank Professor H. Rasiowa for her kind invitation to present these lectures in Warsaw. Thanks are also due to the Stefan Banach International Mathematical Center and the Deutsche Forschungsgemeinschaft for their support.

I gratefully acknowledge Herbert Kuchen's help during the development of this work. He carefully read various drafts and made improving suggestions.

References

[1] L. M. Chirica and D. F. Martin, *An order-algebraic definition of Knuthian semantics*, Math. Systems Theory 13 (1979), 1–27.

[2] B. Courcelles and P. Franchi-Zannettacci, *Attribute grammars and recursive program schemes*, Theor. Comp. Sci. 17 (1982), 163–191 and 235–257.

[3] E. W. Dijkstra, *Recursive Programming*, Num. Math. 2 (1960), 312–318.

[4] C. B. Jones and P. Lucas, *Proving correctness of implementation techniques*, in: *Symposium on Semantics of Algorithmic Languages* (E. Engeler, ed.), Springer Lect. Notes in Math. 188 (1971), 178–211.

[5] H. A. Klaeren, *A constructive method for abstract algebraic software specification*, Theor. Comp. Sci. 30 (1984), 139–204.

[6] R. Milne and Ch. Strachey, *A theory of programming language semantics*, Chapman and Hall, London 1976.

[7] K. Samelson and F. L. Bauer, *Sequentielle Formelübersetzung*, Elektron. Rechenanlagen 1 (1959), 176–182; english translation: Comm. ACM 3 (1960), 76–83.

[8] J. W. Thatcher, E. G. Wagner and J. B. Wright, *More on advice on structuring compilers and proving them correct*, Theor. Comp. Sci. 15 (1981), 223–249.

[9] N. Wirth, *Compilerbau*, Teubner Studienbücher Informatik, 3. Auflage, Stuttgart 1984.

Presented to the semester
Mathematical Problems in Computation Theory
September 16–December 14, 1985