COMPILER GENERATION FROM RELATIONAL SEMANTICS

Mads Dam[1]
Frank Jensen[2]

ABSTRACT

We consider the problem of automatically deriving correct compilers from relational semantic specifications of programming languages. A relational semantics is an assignment of initial-state final-state relations (defined by means of proof rules) to programs. Compilers are generated in three steps. First, the language definition is transformed into a stack semantics in which the storage of semantic values is made explicit. Next, stack rules are assembled into a socalled language scheme containing exactly one rule for each language construct. We consider languages for which non-deterministic branches may be replaced by deterministic ones. Finally, rules are expanded for the purpose of recursion detection, thus obtaining schemes which may be used for code generation in a syntax-directed compiler.

1. Introduction

In this paper we consider relational semantic specifications of programming languages and the problem of deriving correct compilers from such specifications in an automatic way. In relational semantics meaning is given to programs as relations between initial and final states.

The term "relational semantics" seems to have originated with the relational theories of Hoare and Lauer in [Hoare/Lauer 73]. Their semantics, however, have a much less operational flavour than the one considered here, in that they e.g. use invariant properties for the semantics of loops. Our relational semantics is much closer to the deductive systems proposed by Greif and Meyer ([Greif/Meyer 81]) as an alternative to the semantics of Hoare and Lauer. Also Plotkin proposed the use of relational semantics in his work on structural operational semantics ([Plotkin 81]), and in fact our understanding of relational semantics has been heavily influenced by his work.

[1]Authors' address: University of Edinburgh, Dept. of Computer Science, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, U.K.

[2]Authors' address: Aalborg University Centre, Institute of Electronic Systems, Strandvejen 19, 4, DK-9000 Aalborg, Denmark

The basic constituent of a relational semantics is a deductive system (in the style of [Plotkin 81]) consisting of a finite number of axioms and rules of inference, from which formulae denoting valid state transitions may be proved. Formulae in such systems are expressions $\gamma[p]\gamma'$, where γ and γ' are expressions denoting initial and final (or terminal) states respectively, and p is a construct in the programming language under consideration. Axioms and rules of inference are constructed from such formulae together with predicates on states. We shall neither allow quantification nor circular definition of states, and it is from this restriction the operational nature of the relational semantics originates.

Relational semantics in the present setting has been used for several example languages in e.g. [Greif/Meyer 81] and [H. Nielson 84]. It seems that most of the language specification ideas of Plotkin ([Plotkin 81]) are more or less directly transferable to the domain of relational semantics (at least for languages not involving parallellism).

The traditional approach to semantics-directed compiler generation has been to provide a universal compiler, as in e.g. [Mosses 79], [Jones/Schmidt 80], [Jones/Christiansen 81], [Mosses 80] and [Sethi 83] for variants of denotational semantics ([Stoy 77] and [Gordon 79]). This approach is also applicable in our case, noting that a relational semantics is essentially just a collection of first-order sentences in Horn form ([Kowalski 74]). Thus a compiler for Prolog may be considered a universal compiler for relational semantics. This approach, however, has the major drawback that implementation considerations are moved from the domain of language specifications to the (considerably more general) domain of (pure) Prolog programs.

On the other hand works on (non-automated) semantics-directed compiler development usually takes a transformational approach ([Milne/Strachey 76], [Ganzinger 80] and [Wand 82]), that is, compilers are developed through a series of transformations on denotational language definitions involving quite complex correctness proofs. In this paper we apply a similar, but automated, transformational approach for relational semantics.

We shall be concerned mainly with "ordinary" (deterministic) programming languages and the control flow aspects of their implementation.

Three major problems concerning compiler generation are identified. First, the problem of deriving a machine state uniform to all types of program fragments, such that all non-elementary computations can be viewed as a sequential composition of other computations, as defined by the proof rules. This problem is solved by the generation of a stack semantics.

Next, the problem of collecting rules defining the meaning of identical language constructs, and, if possible, replacing non-deterministic branches by deterministic ones. To this end the stack semantics is transformed into a "language scheme", much like a recursive program scheme (cf. [De Bakker/De Roever 72]).

Finally, the problem of detecting recursion in language schemes, such that language scheme rules may be used as schemes for use in code generation. This, however, requires that "definition by syntactical equivalence" is used only to a limited extent.

The approach taken throughout the paper is basically operational - systems are given operational semantics, which are then proven congruent using mainly (numerical) induction.

In section 2 relational semantics is more precisely defined - we shall, due to the subject of this paper, be rather thorough on this point. In sections 3-8 the various transformations on semantics are presented and proved to be correct. As most proofs are entirely routine, they have just been very briefly outlined. Finally, in section 9, some directions for future work are briefly discussed.

2. Relational semantics

The relational semantic metalanguage used in this paper is much like the operational semantics of Plotkin ([Plotkin 81]) with the major difference that all transitions are directly from initial to terminal states.

A language definition will consist of three parts: A definition of the language syntax, a definition of the semantic data types used (e.g. stores, environments), and a finite set of rules, comprising a formal calculus, from which valid state transitions may be deduced.

Programs are considered as abstract trees $A(p_1,\ldots,p_n)$, $n \geq o$, where $p_1,\ldots,p_n$ are programs and A is an operation symbol. To each operation symbol is associated an arity and a sort in the sense of e.g. [ADJ 78], reflecting the fact that programs may be composed of phrases of different semantic significance (e.g. commands, expressions). The set of sorts (which we assume to be finite) is denoted by S, the set of operation symbols by Σ and the set of programs by P. The set Σ is indexed by S*xS (arity and sort), and the set P is indexed by S.

As we are concerned mainly with control flow aspects of compiler generation, we regard states and operations on states as primitive. We shall simply associate sets D_s and D_s' to each sort $s \in S$, namely the initial and terminal states of the sort s, letting $D = \bigcup_{s \in S} (D_s \cup D_s')$.

E.g. for expressions without sideeffects, the initial state might be a mapping of identifiers to values, and the terminal state a value alone. Furthermore, for the purpose of typing, a unique set of (meta-) variable symbols is associated to each program sort and state set. Throughout the paper symbols $\nu,\nu_1\ldots$ are used as program and $\sigma,\sigma_1,\ldots$ as state variables.

Expressions over programs and states are constructed using variables together with elements of sets Ψ,Φ and Π of operation symbols to which an arity and a sort is associated much as for programs above. Operation symbols $\psi \in \Psi$ and $\phi \in \Phi$ denotes functions $\tilde{\psi}$: $P^n \to_p D^m \to_p P$ and $\tilde{\phi}$: $P^n \to_p D^m \to_p D$ respectively and an operation symbol $\pi \in \Pi$ denotes a predicate $\tilde{\pi}$: $P^n \to_p D^m \to_p T$, where $T = \{tt,ff\}$ is the set of thruthvalues. Here $\to_p$ denotes the partial function space constructor and X^n and n-ary cartesian product $X_{s_1} \mathrm{x} \ldots \mathrm{x} X_{s_n}$ for sorts $s_1,\ldots,s_n \in S$. Thus, if $\nu_1,\ldots,\nu_n$ and $\sigma_1,\ldots,\sigma_m$ are program and state variables of appropriate sorts and $\psi \in \Psi$ is a corresponding operation symbol, then $\psi(\nu_1,\ldots,\nu_n)(\sigma_1,\ldots,\sigma_m)$ is an expression of the sort determined by ψ. Notice that, as operation symbols $A \in \Sigma$ may be viewed

as denoting functions $\tilde{A}: P^n \to_p P$, $\Sigma \subseteq \Psi$. Intuitively, an operation symbol $\psi \in (\Psi - \Sigma)$ could denote e.g. a retrieval of a procedure from a state given an identifier, or the construction of a closure from a function and the values of its free variables stored in some state component. The operation $\sim$ mapping operations in Ψ, Φ and Π into their denotations is called an <u>interpretation</u> of Ψ, Φ and Π. The operations in Ψ, Φ and Π will show up in a slightly modified form as primitive instructions in the target programs - thus the compiler generator is actually relative to an interpretation, much like the approach used in [Jones/Christiansen 81].

Now, given a set V of program variables, we may define sets $\Psi(V)$, $\Phi(V)$ and $\Pi(V)$ of "partially evaluated" operations involving only free program variables in V. Thus, given a valuation (an assignment of values to variables) of the variables in V, an operation $\tilde{\psi} \in \Psi(V)$ may be regarded as denoting a function $\tilde{\psi}: D^n \to_p P$ under this valuation. We shall ambiguously abbreviate $\Psi(\emptyset)$, $\Phi(\emptyset)$ and $\Pi(\emptyset)$ by Ψ, Φ and Π respectively - this notation will be used throughout the remainder of the paper.

<u>Definition 1</u>: A <u>formula</u> is an expression $e_1[e_2]e_3$ such that for some $s \in S$, e_1, e_2 and e_3 are expressions over D_s, P_s and D'_s respectively. □

The meaning of a formula $e_1[e_2]e_3$ (relative to some interpretation and valuation) should be fairly obvious: Starting in the state denoted by e_1, by executing the program denoted by e_2 we may end up in the state denoted by e_3 - provided these programs and states are defined. Thus, if $e_1[e_2]e_3$ is actually provable in some calculus $\mathcal{D}$ and - given an interpretation/valuation - p is the program denoted by e_2, d and d' are the states denoted by e_1 and e_3 respectively (assuming they are defined), the pair $\langle d,d' \rangle$ is an element of the transition relation assigned to p by $\mathcal{D}$.

In the definition of proof rules below, we shall distinguish between occurrences of state variable symbols as either <u>defining</u> or <u>applied</u>. Intuitively an occurrence of a variable is defining whenever it is the first occurrence of that particular variable to be instantiated during the proof process (or program execution). The main restriction - and the res-

triction on which the operational nature of our semantics depends - is that every applied occurrence of a variable symbol should be preceded by a defining. Thus we do not allow variables to be circularly defined.

Definition 2: A consequent over program variables V, defined state variables X and applied state variables X' is a formula:

$\sigma[A(\nu_1,\ldots,\nu_n)]\phi(\sigma_1,\ldots,\sigma_m)$, $n\geq o$, $m\geq o$, where

$V = \{\nu_1,\ldots,\nu_n\}$, $\nu_i \neq \nu_j$ when $i\neq j$, $X=\{\sigma\}$, $X'=\{\sigma_1,\ldots,\sigma_m\}$,

$\phi\in\Phi(V)$ and $A\in\Sigma$. □

Definition 3: An antecedent over program variables V, defined state variables X and applied state variables X' is either a test formula:

$\pi(\sigma_1,\ldots,\sigma_n)$, $n\geq o$, where

$\pi\in\Pi(V)$, $X=\emptyset$ and $X'=\{\sigma_1,\ldots,\sigma_n\}$ or a (transition-) formula:

$\phi(\sigma_1,\ldots,\sigma_n)[\psi(\sigma'_1,\ldots,\sigma'_m)]\sigma$, $n\geq o$, $m\geq o$, where

$\phi\in\Phi(V)$, $\psi\in\Psi(V)$, $X=\{\sigma\}$ and $X'=\{\sigma_1,\ldots,\sigma_n,\sigma'_1,\ldots,\sigma'_m\}$. □

Definition 4: A proof rule is an expression:

$$\frac{F_1,\ldots,F_n}{F}\ ,\ n\geq o,$$

where F is a consequent over program variables V, defined state variables X and applied state variables X', and for all $i, 1\leq i\leq n$, F_i is an antecedent over program variables V, defined state variables X_i and applied state variables X'_i. It is required first, that $X'\subseteq X\cup X_1\cup\ldots\cup X_n$ and $X'_i\subseteq X\cup X_1\cup\ldots\cup X_{i-1}$, $1\leq i\leq n$, and secondly, that $X,X_1,\ldots,X_n$ are mutually disjoint. □

If, in definition 4, n=o, the rule is called an axiom, otherwise it is an inference rule. The second restriction is merely a convenience introduced to avoid implicit testing of variables.

Intuitively a proof rule describes a way to compute a terminal state from an initial state on a certain program construct. This computation may involve computations of other initial and terminal states using other programs or it may involve tests on programs or states. Notice that - as opposed to e.g. denotational semantics - meaning need not be given to programs in terms of their immediate constituents (in case of e.g. loops it cannot). Notice also that according to definition 4 it should be possible to "execute" all antecedents in the order they are written. This need not necessarily be the case. One might employ data dependencies among antecedents to detect sequencing constraints and then use these as the point of departure for compiler generation.

Example 1: Consider a small programming language with $S=\{Exp,Cmd\}$, $\Sigma_{\varepsilon,Exp}=\{Ide\}$, $\Sigma_{ExpExp,Exp}=\{Op\}$, $\Sigma_{Exp,Cmd}=\{Ide:=\}$, $\Sigma_{CmdCmd,Cmd}=\{Seq\}$, $\Sigma_{ExpCmd,Cmd}=\{WhileDo\}$ and $V_{Exp}=\{e,e_1,\ldots\}$, $V_{Cmd}=\{c,c_1,\ldots\}$. Let $D_{Exp}=D_{Cmd}=D'_{Cmd}=Store$, $D'_{Exp}=Value$, $X_{Store}=\{\sigma,\sigma_1,\ldots\}$, $X_{Value}=\{\nu,\nu_1,\ldots\}$ and operations in Ψ, Φ and Π be as obvious from the rules:

X1: $\sigma[Ide]\ val_{Ide}(\sigma)$

X2: $$\frac{\sigma[e_1]\nu_1,\ \sigma[e_2]\nu_2}{\sigma[Op(e_1,e_2)]f(\nu_1,\nu_2)}$$

where f is an operation corresponding to Op

X3: $$\frac{\sigma[e]\nu}{\sigma[Ide:=(e)]update_{Ide}(\sigma,\nu)}$$

X4: $$\frac{\sigma[c_1]\sigma',\sigma'[c_2]\sigma''}{\sigma[\mathrm{Seq}(c_1,c_2)]\sigma''}$$

X5: $$\frac{\sigma[e]\nu,\mathrm{Is}_{ff}(\nu)}{\sigma[\mathrm{WhileDo}(e,c)]\sigma}$$

X6: $$\frac{\sigma[e]\nu,\mathrm{Is}_{tt}(\nu),\sigma[c]\sigma',\sigma'[\mathrm{WhileDo}(e,c)]\sigma''}{\sigma[\mathrm{WhileDo}(e,c)]\sigma''}$$ □

A formula F is said to be provable in (the deductive system of) a language definition $\mathcal{D}$ (written $\vdash_{\mathcal{D}} F$) if and only if there exists a proof tree for F in $\mathcal{D}$, proof trees being defined inductively as follows:
First, if F is an instance of an axiom A in $\mathcal{D}$ (that is, the value of A under some interpretation and valuation) or it is the value tt, then F is a proof tree for F in $\mathcal{D}$ (of height o).
Next, if F', $F_1',\ldots,F_n'$ are instances of formulae F, $F_1,\ldots,F_n$ respectively (that is, they are instances under identical interpretations/valuations) and $T_1,\ldots,T_n$ are proof trees for $F_1',\ldots,F_n'$ respectively in $\mathcal{D}$, then, provided $\frac{F_1,\ldots,F_n}{F}$ is an inference rule in $\mathcal{D}$, $\frac{T_1,\ldots,T_n}{F}$ is a proof tree for F' in $\mathcal{D}$ (of height $\max(\mathrm{height}(T_1),\ldots,\mathrm{height}(T_n))+1$). We shall denote by $\mathrm{Tr}(\mathcal{D})$ the set of provable transition formulae in $\mathcal{D}$.

This semantics of language definitions is basically non-operational, due to its inherent non-determinacy. In the following sections the semantics is made gradually more operational through various transformations on language definitions to the point where they may be represented as target programs on a fixed, abstract machine.

3. Stack semantics

The first step towards compiler generation consists of making the storage of state values implicit in language definitions explicit by computing them on a stack. Through this, the types of initial and terminal states are made uniform to

all program sorts and thus in the stack semantics all programs and operations $\phi \in \Phi$ may be viewed as transformers of a state of a single type.

We shall need the following operations on stacks:

- nil denotes the empty stack (occasionally we shall use nil with a subscript to indicate the type of the stack)
- $d:\xi$ denotes ξ with d added on top
- $pop_k(\xi)$ denotes ξ with the top k elements removed
- $top_k(\xi)$ denotes the k'th element of ξ from top
- $|\xi|$ denotes the height of ξ
- $\xi_1 \S \xi_2$ denotes the concatenation of ξ_1 and ξ_2
- $pos(d,\xi)$ denotes the least k s.t. $top_k(\xi)=d$, if such a k exists.

In expressions all applied occurrences of state variables are replaced by references to positions on a stack. To this end a stack environment $u:X \to_p N$, where X is the set of state variables and N the set of natural numbers, is used.

Let () denote the everywhere undefined function, $[a \to b]f$ the function mapping a to b and elsewhere behaving like f, and next(f), where the range of f is N, the function $\lambda a.f(a)+1$.

Let $r = \dfrac{F_1,\ldots,F_n}{F_o}$ be a proof rule. Then the stack environments $env(F_i), o \le i \le n$, are defined as (assuming X_i is the set of defined variables for F_i):

- Let $X_o=\{\sigma\}$. Then $env(F_o)=[\sigma \to 1]()$.
- If $X_i=\{\sigma\}, 1 \le i \le n$, then $env(F_i)=[\sigma \to 1]next(env(F_{i-1}))$.
- If $X_i=\emptyset, 1 \le i \le n$, then $env(F_i)=env(F_{i-1})$.

Now, let e be an expression, u a stack environment s.t. $u(\sigma)$ is defined for all free occurrences of state variables σ in e, and ξ a variable over the set $\Xi=D^*$ of stacks. Then $stack(e,u,\hat{e})$ is the expression e with all free occurrences of state variables σ replaced by the expression $top_i(\hat{e})$, where $i=u(\sigma)$.

<u>Definition 5</u>: Let $r = \frac{F_1,\dots,F_n}{F_o}$, $n \geq o$, be a proof rule, and for all $i, o \leq i \leq n$, k_i be the number of transition formulae among $F_1,\dots,F_i$, $\{\xi_o,\xi_1,\dots\}$ a fixed set of variable symbols over Ξ. Then <u>the stack rule generated by r</u> is a rule $\hat{r} = \frac{\hat{F}_1,\dots,\hat{F}_n}{\hat{F}_o}$, where

- if $F_o = e_o[e_o']e_o''$, then $\hat{F}_o =$

  $\xi_o[e_o']stack(e_o'',env(F_n),\xi_{k_n}):pop_{k_n+1}(\xi_{k_n})$,

- if F_i, $1 \leq i \leq n$, is a test formula e_i, then

  $\hat{F}_i = stack(e_i,env(F_i),\xi_{k_i})$, and

- if F_i, $1 \leq i \leq n$, is a transition formula $e_i[e_i']e_i''$, then

  $\hat{F}_i = \hat{e}_i[stack(e_i',next(env(F_{i-1})),\hat{e}_i)]\xi_{k_i}$, where

  $\hat{e}_i = stack(e_i,env(F_{i-1}),\xi_{k_{i-1}}):\xi_{k_{i-1}}$. □

Actually, if, in definition 5, $e_o=\sigma$ and σ is a variable over, say D_s, an antecedent $top_1(\xi_o) \in D_s$ should be added to $\hat{r}$ in order to prevent type conflicts otherwise trapped in the language definition. We shall, however, assume this change to be made implicitly to all stack rules.

<u>Example 2</u>: Consider rule X2 of ex.1. The stack rule generated by X2 is:

$$\frac{top_1(\xi_o):\xi_o[e_1]\xi_1,top_2(\xi_1):\xi_1[e_2]\xi_2}{\xi_o[Op(e_1,e_2)]f(top_2(\xi_2),top_1(\xi_2)):pop_3(\xi_2)}$$ □

If $\mathcal{D}$ is a language definition, then Stack($\mathcal{D}$) denotes the language definition consisting of all stack rules generated by rules in $\mathcal{D}$.

It is easily verified that Stack($\mathcal{D}$) is actually well-defined - i.e. that, according to definitions 2,3 and 4, the stack rule generated from a proof rule is itself defined and a proof rule.

Inspecting definition 5, we see that a stack rule $\hat{r} = \frac{\hat{F}_1,\ldots,\hat{F}_n}{\hat{F}_o}$ has the following properties:

- $\hat{F}_o$ has the form $\xi_o[A(\nu_1,\ldots,\nu_l)]\phi_o(\xi_m)$, where m is the number of transition formulae among $\hat{F}_1,\ldots,\hat{F}_n$, and
- $\hat{F}_i, 1 \leq i \leq n$, has either the form $\phi_i(\xi_j)[\psi_i(\phi_i(\xi_j))]\xi_{j+1}$ or the form $\pi_i(\xi_j)$, where j is the number of transition formulae among $\hat{F}_1,\ldots,\hat{F}_{i-1}$.

A language definition in which all proof rules possess these properties is said to be in stack form.

4. Language schemes

In the stack semantics we may distinguish three kinds of "actions": Transformations of states defined by programs (Ψ), primitive state transformations (Φ) and primitive tests (Π) - and operations on actions corresponding to sequencing and non-deterministic branching, the latter originating from the possibility of existence of different rules on identical language constructs. In language schemes the set of rules in the stack semantics defining the meaning of a particular language construct $A(\nu_1,\ldots,\nu_n)$ gives rise to a single production rule $A(\nu_1,\ldots,\nu_n)\rightarrow h$, where h is an action to be more formally defined below, called a right hand side of the language scheme considered.

Language schemes are semantic descriptions on much the same level of abstraction as the store semantics of [Milne/Strachey 76]. In store semantics meaning is given to programs in terms

of command continuations alone, in analogy to the uniform type of transition relations obtained here by the stack semantics. In fact language schemes adds nothing new to the stack semantics in terms of meaning - it is merely introduced as a convenience for the subsequent generation of code schemes.

Now define the set H of <u>actions</u> by the grammar:

$$h ::= h \cdot h \mid h \cup h \mid \pi \to h, h \mid \alpha$$

$$\alpha ::= \psi \mid \phi \mid \pi \mid \underline{\text{error}} \mid \varepsilon$$

where <u>error</u> and ε are fixed symbols, $h \in H$, $\psi \in \Psi(V)$, $\phi \in \Phi(V)$ and $\pi \in \Pi(V)$ for some set V of program variables. An action of the form α is called <u>atomic</u>, and an atomic action not in $\Pi(V)$ (for any V) is called <u>non-predicate</u>.

Intuitively $\cdot$ denotes sequencing, $\cup$ denotes non-deterministic and $\to$ denotes deterministic branching, <u>error</u> denotes an error (in-)action and ε denotes the empty action. We shall assume $\cdot$ to have the strongest binding power and otherwise employ parentheses to disambiguate.

The semantics of language schemes is given operationally in the style of [Plotkin 81] relative to a set D of states and an interpretation $\sim$ of Ψ, Φ and Π (both D and $\sim$ obtained from the corresponding stack semantics - we therefore only have to consider operations in Ψ, Φ and Π on single states).

Letting L denote the set of language schemes, we define the transition system $T_l = \langle \Gamma_l, \Rightarrow_l \rangle$, where $\gamma \in \Gamma_l = D \times H + D$ is the set of <u>configurations</u>, and $\Rightarrow_l \subseteq L \times ((\Gamma_l - D) \times \Gamma_l)$ is the transition relation defined by the deductive system Δ_l below.

We use the notation $l \triangleright \gamma \Rightarrow \gamma'$ for an element $(l, (\gamma, \gamma'))$ of $\Rightarrow_l$ and letting $\Rightarrow_l^*$ denote the reflexive and transitive closure of $\Rightarrow_l$.

Also we shall use the notation $\bar{\nu}$ and $\bar{p}$ as abbreviations of sequences $\nu_1, \ldots, \nu_n$ and $p_1, \ldots, p_m$, and, where $n=m$, all ν_i, $1 \leq i \leq n$, distinct and p_i an instance of ν_i, $h[\bar{p}/\bar{\nu}]$ as an abbreviation of the expression obtained by (simultaneous) substitution of the variable ν_i by p_i in h.

Now Δ_l consists of rules:

E_l: $\quad l \triangleright \langle \sigma, \varepsilon \rangle \Rightarrow_l \sigma$,

PHI_l: $\quad l \triangleright \langle \sigma, \phi \rangle \Rightarrow_l \tilde{\phi}(\sigma)$, if $\phi \in \Phi$,

PL_l: $\quad \dfrac{\tilde{\pi}(\sigma)}{l \triangleright \langle \sigma, \pi \rangle \Rightarrow_l \sigma}$, if $\pi \in \Pi$,

$PSI1_l$: $\quad l \triangleright \langle \sigma, \psi \rangle \Rightarrow_l \langle \sigma, \tilde{\psi}(\sigma) \rangle$, if $\psi \in \Psi - P$,

$PSI2_l$: $\quad l \triangleright \langle \sigma, p \rangle \Rightarrow_l \langle \sigma, h[\bar{p}/\bar{v}] \rangle$, if $p \in P$, $p = A(\bar{p})$ and

$(A(\bar{v}) \to h) \in l$.

$CP1_l$: $\quad \dfrac{l \triangleright \langle \sigma, h_1 \rangle \Rightarrow_l \langle \sigma', h_1' \rangle}{l \triangleright \langle \sigma, h_1 \cdot h_2 \rangle \Rightarrow_l \langle \sigma', h_1' \cdot h_2 \rangle}$

$CP2_l$: $\quad \dfrac{l \triangleright \langle \sigma, h_1 \rangle \Rightarrow_l \sigma'}{l \triangleright \langle \sigma, h_1 \cdot h_2 \rangle \Rightarrow_l \langle \sigma', h_2 \rangle}$

$BR1_l$: $\quad l \triangleright \langle \sigma, h_1 \cup h_2 \rangle \Rightarrow_l \langle \sigma, h_1 \rangle$

$BR2_l$: $\quad l \triangleright \langle \sigma, h_1 \cup h_2 \rangle \Rightarrow_l \langle \sigma, h_2 \rangle$

$TST1_l$: $\quad \dfrac{\tilde{\pi}(\sigma)}{l \triangleright \langle \sigma, \pi \to h_1, h_2 \rangle \Rightarrow_l \langle \sigma, h_1 \rangle}$, if $\pi \in \Pi$,

$TST2_l$: $\quad \dfrac{\sim\tilde{\pi}(\sigma)}{l \triangleright \langle \sigma, \pi \to h_1, h_2 \rangle \Rightarrow_l \langle \sigma, h_2 \rangle}$, if $\pi \in \Pi$.

The *behaviour of* T_l is defined to be a mapping $BT_l: L \to H \to 2^{D \times D}$ such that for all $l \in L$, $h \in H$, $d, d' \in D$, $\langle d, d' \rangle \in BT_l(l)(h)$ if and only if $\vdash_{\Delta_l} l \triangleright \langle d, h \rangle \Rightarrow_l^{*} d'$. Of course we are particularly interested in the behaviour of T_l on actions $p \in P$ - that is, programs without free program variables. This explains why no rules are present for actions of the form $v \in V$. Notice also that no rules are present for *error*-actions as should be expected.

Our definition of behaviour leads us to define an equivalence relation $\equiv$ such that for all $h_1, h_2 \in H$, $h_1 \equiv h_2$ if and

only if for all $l \in L$ $BT_l(l)(h_1)=BT_l(l)(h_2)$ - i.e. if they behave the same, independently of the particular language scheme.

It is easy to see that $\equiv$ is in fact a congruence relation. First it is useful to state (letting $\circ$ denote relational composition):

Lemma 1: For all $l \in L$, $h_1,h_2 \in H$:

$$BT_l(l)(h_1 \cdot h_2) = BT_l(l)(h_1) \circ BT_l(l)(h_2).$$

Proof: Immediate from the definition of BT_l and Δ_l. □

Lemma 2: For all $h_1,h_1',h_2,h_2' \in H$ and $\pi \in \Pi$, if $h_1 \equiv h_1'$ and $h_2 \equiv h_2'$ then:

a) $h_1 \cdot h_2 \equiv h_1' \cdot h_2'$,

b) $h_1 \cup h_2 \equiv h_1' \cup h_2'$,

c) $\pi \to h_1, h_2 \equiv \pi \to h_1', h_2'$.

Proof: a) from lemma 1, b) from $BR1_l$, $BR2_l$, c) from $TST1_l$, $TST2_l$. □

Thus the set $H/\equiv$ of actions modulo behavioural equivalence is actually well-defined with operations $\cdot$ and $\cup$ defined by $[h_1]\cdot[h_2] =_D [h_1 \cdot h_2]$ and $[h_1]\cup[h_2] =_D [h_1 \cup h_2]$, where $[h]$ denotes the equivalence class of $\equiv$ containing h.

Proposition 1: The structure $\langle H/\equiv, \cdot, \cup, [\varepsilon], [\underline{error}] \rangle$ has the properties:

a) $\langle H/\equiv, \cdot, [\varepsilon] \rangle$ is a monoid,

b) $\langle H/\equiv, \cup, [\underline{error}] \rangle$ is a commutative monoid, and

c) The operation $\cdot$ distributes over $\cup$.

Proof: By lemma 1 and Δ_l. □

Now consider a language definition $\mathcal{D}$ in stack form. Let Rules($\mathcal{D}$,A) denote the set of rules in $\mathcal{D}$ defining the program operator $A \in \Sigma$. The language scheme corresponding to $\mathcal{D}$ is generated quite easily using the mappings $\mathcal{A}$ and $\mathcal{R}$ of antecedents and rules in definitions in stack form into actions defined by:

$$\mathcal{A}(\phi(\xi)[\psi(\phi(\xi))]\xi')=\phi\cdot\psi, \quad \mathcal{A}(\pi(\xi))=\pi,$$

and, if $r = \frac{F_1,\ldots,F_n}{F}$, where F has the form $\xi[A(\bar{v})]\phi(\xi')$, then:

$$\mathcal{R}(r)=\mathcal{A}(F_1)\cdot\ldots\cdot\mathcal{A}(F_n)\cdot\phi.$$

If Rules($\mathcal{D}$,A)=$\{r_1,\ldots,r_m\}$ then the language scheme generated from $\mathcal{D}$ contains the production:

$$A(\bar{v}) \rightarrow \begin{cases} \mathcal{R}(r_1)\cup\ldots\cup\mathcal{R}(r_m), & \text{if } m\geq 1 \\ \underline{\text{error}}, & \text{if } m=0 \end{cases}$$

where $A(\bar{v})$ is the program structure assumed common to all consequents of rules $r_1,\ldots,r_m$.

Example 3: Consider a language with sorts Exp, Cmd, operations $\Sigma_{\varepsilon,Cmd}$={Skip}, $\Sigma_{CmdCmd,Cmd}$={Seq} and $\Sigma_{ExpCmd,Cmd}$={WhileDo}. Let D denote a primitive set of states and let $D_{Exp}=D_{Cmd}=D'_{Exp}=D'_{Cmd}=D$.
The rules are:

X31: $\sigma[\text{Skip}]\sigma$

X32: $\dfrac{\sigma[c_1]\sigma',\sigma'[c_2]\sigma''}{\sigma[\text{Seq}(c_1,c_2)]\sigma''}$

X33: $\dfrac{\sigma[e]\sigma',\text{isTrue}(\sigma'),\sigma'[\text{Seq}(c,\text{WhileDo}(e,c))]\sigma''}{\sigma[\text{WhileDo}(e,c)]\sigma''}$

X34: $\dfrac{\sigma[e]\sigma',\text{isFalse}(\sigma')}{\sigma[\text{WhileDo}(e,c)]\sigma'}$

This definition is obviously in stack form. Letting I denote the identity on D the following language scheme is obtained:

$$\mathrm{Skip} \rightarrow I,$$

$$\mathrm{Seq}(c_1,c_2) \rightarrow I \cdot c_1 \cdot I \cdot c_2 \cdot I,$$

$$\mathrm{WhileDo}(e,c) \rightarrow I \cdot e \cdot \mathrm{isTrue} \cdot I \cdot \mathrm{Seq}(c,\mathrm{WhileDo}(e,c)) \cdot I \cup$$

$$I \cdot e \cdot \mathrm{isFalse} \cdot I$$

Notice that as $I \equiv \varepsilon$ almost all occurrences of I in this language scheme may be deleted. □

Theorem 1, (Correctness of language scheme generation): Let $\mathcal{D}$ be an arbitrary language definition in stack form, and l the language scheme generated from $\mathcal{D}$. Then for all $p \in P$ and $d,d' \in D$, $\vdash_{\mathcal{D}} d[p]d'$ if and only if $\langle d,d' \rangle \in BT_1(l)(p)$.

Proof: ⇒ is proved by induction in the height of proof trees, and ⇐ by induction in the length n of the derivation $l \triangleright \langle d,p \rangle \Rightarrow_l^n d'$. □

5. Simple language schemes

Language schemes may be implemented in a variety of ways - e.g. using some kind of backtracking evaluation strategy. This, however, will obviously couse correctness problems due to non-determinism, so to ensure correctness we have somehow to restrict the class of language definitions considered.

We shall choose to consider only a (natural, we think) class of language schemes for which all non-deterministic branches may be replaced by deterministic ones, using the simple transformation algorithm presented below.

First, notice that in any production $A(\bar{v}) \rightarrow h$ in a language scheme $l \in L$ generated from an arbitrary language definition in stack form, h will have the form $h=h_1 \cup \ldots \cup h_n$, $n \geq 1$, where each h_i, $1 \leq i \leq n$, is a product $(\alpha_{i,1} \cdot \ldots \cdot \alpha_{i,m_i})$ of atomic actions. If in a language scheme $l \in L$ all right hand sides have the above form, then l is said to be normal, and a language

scheme in which no right hand side contains non-deterministic branches - i.e. actions are generated by the grammar:

$$h ::= h\cdot h \mid \pi\rightarrow h,h \mid \alpha$$

$$\alpha ::= \phi \mid \psi \mid \underline{\text{error}} \mid \varepsilon$$

- is called simple. We shall use the notations H_s, L_s and T_s when we restrict H, L and T_1 to simple language schemes.

In order to generate simple language schemes we must have some means of looking into the internal structure of predicates $\pi\in\Pi(V)$ - we shall simply assume a negation sign $\neg$ to be available in constructing predicates, so that if $\pi\in\Pi(V)$, then $\neg\pi\in\Pi(V)$.

Consider a normal language scheme $l\in L$. The simple language scheme $l'\in L_s$ generated by l is - if it is defined - obtained from l by applying the (partial) mapping H defined below to all right hand sides of l.

The mapping H is defined inductively by:

a) If h is a non-predicate atomic action α, then $H(h)=\alpha$.

b) If $h=h_1\cup\ldots\cup h_n$, $n\geq 1$, and there exists a non-predicate atomic action α and $h'_1,\ldots,h'_n\in H$ such that $h_i=\alpha\cdot h'_i$, $1\leq i\leq n$, then $H(h)=\alpha\cdot H(h'_1\cup\ldots\cup h'_n)$.

c) If $h=h_1\cup\ldots\cup h_n$, $n\geq 1$, and there exists $\pi\in\Pi(V)$ and $h'_1,\ldots,h'_n\in H$ such that either $h_i=\pi\cdot h'_i$ or $h_i=\neg\pi\cdot h'_i$, $1\leq i\leq n$, then $H(h)=\pi\rightarrow H(\cup\{h'_i \mid h_i=\pi\cdot h'_i, 1\leq i\leq n\}),$
$H(\cup\{h'_i \mid h_i=\neg\pi\cdot h'_i, 1\leq i\leq n\})$.

The notation $\cup\{h_1,\ldots,h_n\}$ is defined by: If $n\geq 1$, then $\cup\{h_1,\ldots,h_n\}=h_1\cup\ldots\cup h_n$ and, for n=o, $\cup\emptyset=\underline{\text{error}}$.

Intuitively a simple language scheme may be generated from a language definition D if and only if, for any two rules of D defining identical language constructs, all computations of those rules are identical up to a point where one rule contains a test and the other contains either the same or the negation of this test.

Example 4: Consider the production for WhileDo of ex.3. The production of the corresponding simple language scheme is:

$$\mathrm{WhileDo}(e,c) \rightarrow I \cdot e \cdot \mathrm{isTrue} \rightarrow I \cdot \mathrm{Seq}(c, \mathrm{WhileDo}(e,c)) \cdot I, I$$

assuming isFalse=$\neg$isTrue. □

6. Expansion trees

Compilers are generated using productions as schemes for use in code generation, mapping operations on actions into operations on code. First, however, recursion must be eliminated - to which end the class of language schemes considered has to be somewhat restricted. Recursion is detected through the repeated replacement of actions of the form $A(\overline{p})$ by $h[\overline{p}/\overline{v}]$, where $A(\overline{v}) \rightarrow h$ is a production in the simple language scheme considered, until all such actions have been expanded once.

Now, the set E of expansion trees is defined inductively by:

- $\cdot\varepsilon$, $\cdot$error, $\cdot\phi$, $\cdot\psi$ are expansion trees for all $\phi \in \Phi(V)$, $\psi \in \Psi(V)$.
- If $e, e_1, e_2 \in E$ then $\overset{A(\overline{p})}{\underset{e}{|}}$, $e_1 \wedge e_2$, $e_1 \overset{\pi}{\wedge} e_2$ are expansion trees for all $A(\overline{p}) \in P(V)$, $\pi \in \Pi(V)$.

Let $\mathcal{E}$ be the obvious mapping from H_s into E (with $\mathcal{E}(\psi) = \cdot\psi$ for all $\psi \in \Psi(V)$), and $\mathcal{E}'$ the (almost) equally as obvious mapping representing expansion trees as simple actions - note only that $\mathcal{E}'(\overset{A(\overline{p})}{\underset{e}{|}}) = \mathcal{E}'(e)$. Of course for all $h \in H_s$, $h = \mathcal{E}'(\mathcal{E}(h))$.

Definition 6: Let $l \in L_s$ and $e \in E$. If there exists a leaf of e labelled $A(\overline{p})$, $A(\overline{p}) \in P(V)$, then, if $A(\overline{v}) \rightarrow h$ is a production in l and the path from root to leaf contains no other node labelled $A(\overline{p})$, the tree obtained from e by replacing this leaf by the tree $\overset{A(\overline{p})}{\underset{\mathcal{E}(h[\overline{p}/\overline{v}])}{|}}$ is called an expansion of e. □

Proposition 2: For all $e, e' \in E$ such that e' is an expansion of e, $\mathcal{E}'(e) \equiv \mathcal{E}'(e')$.

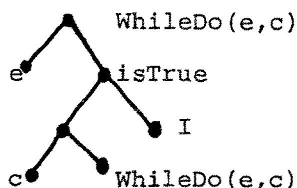
Proof: The result follows from the definition of $\mathcal{E}'$, if we can prove $A(\overline{p}) \equiv h[\overline{p}/\overline{v}]$. But this is easy from rules $PSI1_1$ and $PSI2_1$ in Δ_1. □

Initially productions $A(\overline{v}) \to h$, are represented as expansion trees $\overset{A(\overline{v})}{\underset{\mathcal{E}(h)}{|}}$. The expansion tree generated from the production $A(\overline{v}) \to h$, if it exists, is obtained from this initial tree by applying to it a sequence of expansions until the point where no more expansions can be made.

To ensure that expansion trees in fact do exist, we shall enforce the following expandability condition on all simple language schemes considered:
At no point in the expansion process of any production in l shall a leaf $A(\overline{p})$, such that the path from root to leaf contains another node labelled $A(\overline{p}')$, $A(\overline{p}), A(\overline{p}') \in P(V)$ be expanded.

As all language schemes contains only a finite number of productions, and thus only a finite number of operations $A \in \Sigma$, it is clear that the expandability condition is actually sufficient to ensure termination, and furthermore it follows directly from proposition 2 that, for any expandable production $A(\overline{v}) \to h$, $h \equiv \mathcal{E}'(e)$, where e is the (well-defined) expansion tree generated from $A(\overline{v}) \to h$.

Example 5: The expansion tree generated from the WhileDo production of ex. 4. is:



omitting most of the I's. As an example of a nonexpandable language scheme consider:

WhileDo(e,c)→e·isTrue→RepeatUntil(c,Not(e)),I

RepeatUntil(c,e)→c·WhileDo(Not(e),c). □

For all expandable $l \in L_s$ we let Forest(l) denote the set of all expansion trees generated from productions in l.

It is convenient to represent such forests as mappings $F(l) \in F = P \rightarrow_p E \times W$, where $W = V \rightarrow_f P$ is the set of valuations, such that if e is an expansion tree generated from a production $A(\nu) \rightarrow h$ in l, then for all p of the form $A(\overline{p})$, $F(l)(p) = \langle e,w \rangle$, where for any v_i, p_i in $\overline{\nu}, \overline{p}$, $w(\nu_i) = p_i$.

7. Compiler generation

Now compilers are generated using the expansion trees as schemes for code generation. Before delving into the compiler generation algorithm, however, a question arises concerning the handling of free program variables in operations ψ, ϕ and π. As we would like to replace operations ψ, ϕ and π containing free program variables by "equivalent" operations on label values in the target language, we must require these operations to handle program values in a disciplined way. This is done easily (and - we believe - in accordance with intuition) for operations in $\psi \in \Psi(V) - P(V)$ and $\pi \in \Pi(V)$ by simply not allowing them to contain free program variables whatsoever. The question for operations $\phi \in \Phi(V)$ is more difficult, as they are the operations actually binding and manipulating programs in the state. Intuitively we shall require from all operations $\phi \in \Phi(V)$ that they do not construct, test or dismantle programs in any way - i.e. they are only able to store them.

To formalize this, notice first that for any state $d' \in D$ reachable from an initial state $d \in D$ (that is, $\vdash_{\Delta_1} l \triangleright \langle d,p \rangle \Rightarrow_l^* \langle d',h \rangle$ for some $l \in L_s$, $p \in P$ and $h \in H_s$) it will be the case that $d' = \tilde{\phi}_1(\ldots(\tilde{\phi}_n(d))\ldots)$ for some $n \geq o$ and $\phi_1, \ldots, \phi_n \in \Phi$. Furthermore, it is clear that any reachable state $d \in D$ may be written $d = \tilde{\phi}_1(\ldots(\tilde{\phi}_n(\tilde{\delta}))\ldots)$ for some constant symbol δ over D, $n \geq o$ and $\phi_1, \ldots, \phi_n \in \Phi$.

Now consider an expression $\phi_1(\ldots(\phi_n(\delta))\ldots)$ with $n \geq o$ and V the set of (program) variables free in $\phi_1,\ldots,\phi_n,\delta$. Let w denote an arbitrary valuation. Then for any operation $\psi \in \Psi - P$ such that $p=\tilde{\psi}(\tilde{\phi}_1(\ldots(\tilde{\phi}_n(\tilde{\delta}))\ldots)$ is defined it should be the case that $p=w(\nu)$ for some $\nu \in V$. If this condition is satisfied by all interpretations $\sim$ it seems reasonable to assume, that corresponding operations $\psi_1,\phi_{1,1}\ldots, \phi_{1,n},\delta_1$ on label values in fact do exist such that - for any "label valuation" $w_1:V \to_f$ Labels such that $w_1(\nu)$ is defined if and only if $w(\nu)$ is defined - $\tilde{\psi}_1(\tilde{\phi}_{1,1}(\ldots(\tilde{\phi}_{1,n}(\tilde{\delta}_1))\ldots))=w_1(\nu)$ if and only if $\tilde{\psi}(\tilde{\phi}_1(\ldots(\tilde{\phi}_n(\tilde{\delta}))\ldots))=w(\nu)$. Let Ψ', Φ' denote the sets of such "label operations" corresponding to $\Psi-P,\Phi$.
Target programs $\iota \in I$ are defined by:

$\iota ::= \iota;\iota \mid \pi \to \iota,\iota \mid$ <u>call</u> $1 \mid \phi \mid$ <u>exec</u> $\psi \mid$ <u>return</u> $\mid$ <u>error</u>,

where $\pi \in \Pi$, $\phi \in \Phi'$, $\psi \in \Psi'$ and $1 \in$labels. The structuring of target programs present here is easily eliminated using associativeness and replacing target programs in branches by labels. This, however, is quite inessential to the present treatment and will thus be omitted.

Target programs are interpreted relative to an environment binding labels to programs and a stack of programs remaining to be executed (a control stack). Let $T_t=\langle\Gamma_t,\Rightarrow_t\rangle$, where:

$\gamma \in \Gamma_t = D \times C + D$	(configurations),
$c \in C = I^*$	(controls),
$\Rightarrow_t \subseteq U \times ((D \times I^*) \times \Gamma_t)$	(the transition relation), and
$u \in U = \text{labels} \to_f I$	(environments).

The transition relation is defined by Δ_t:

NIL_t : $u \triangleright \langle\sigma, nil_c\rangle \Rightarrow_t \sigma$

$CALL_t$: $u \triangleright \langle\sigma,$ <u>call</u> $1:c\rangle \Rightarrow_t \langle\sigma, u(1):c\rangle$

PHI_t : $u \triangleright \langle\sigma, \phi:c\rangle \Rightarrow_t \langle\tilde{\phi}(\sigma), c\rangle$

$EXEC_t$: $u \blacktriangleright \langle\sigma, \underline{exec}\ \psi : c\rangle \Rightarrow_t \langle\sigma, u(\tilde{\psi}(\sigma)) : c\rangle$

RET_t : $u \blacktriangleright \langle\sigma, \underline{return} : c\rangle \Rightarrow_t \langle\sigma, c\rangle$

CP_t : $u \blacktriangleright \langle\sigma, (c_1 ; c_2) : c\rangle \Rightarrow_t \langle\sigma, c_1 : c_2 : c\rangle$

TST_t : $u \blacktriangleright \langle\sigma, (\pi \rightarrow c_1, c_2) : c\rangle \Rightarrow_t \langle\sigma, c' : c\rangle$, where, if $\tilde{\pi}(\sigma)=tt$ then $c'=c_1$ and if $\tilde{\pi}(\sigma)=ff$ then $c'=c_2$.

As before the $\overline{\underline{\text{behaviour of } T_t}}$ is a mapping $BT_t : U \rightarrow I \rightarrow 2^{D \times D}$ such that for all $\overline{u \in U, \iota \in I \text{ and } d, d' \in D}$, $\langle d, d'\rangle \in BT_t(u)(\iota)$ if and only if $\vdash_{\Delta_t} u \blacktriangleright \langle d, \iota : nil_c\rangle \Rightarrow_t^* d'$.

The compiler generator is a mapping $cg : F \rightarrow_p P \rightarrow I$ taking a forest of expansion trees in F, a source program and yielding a target program (and with it an environment $u \in U$). It is defined using the mapping $C : F \times E \times W \rightarrow I$ by $cg(f)(p) = C(f, f(p){\downarrow}1, f(p){\downarrow}2)$. The mapping C is defined by cases on the tree $e \in E$, and noting, that the only target programs to be bound in the environment are those obtained from trees representing programs $p \in P$, we set labels=P.

Now the mapping C is defined by (for simplicity describing the elaboration of environments $u \in U$ through side effects):

$C(f, \cdot\varepsilon, w) = \underline{return}$,

$C(f, \cdot\underline{error}, w) = \underline{error}$,

$C(f, \cdot\phi, w) = \phi'$, where $\phi' = \phi[w]$ - i.e. ϕ with all free variables ν in ϕ replaced by $w(\nu)$. This case gives for all such ν rise to the bindings $u(w(\nu)) = cg(f)(w(\nu))$ in u,

$C(f, \cdot\psi, w) = \underline{exec}\ \psi$, if $\psi \in \Psi - P$,

$C(f, \cdot\nu, w) = cg(f)(w(\nu))$,

$C(f, \cdot A(\overline{p}), w) = \underline{call}\ A(\overline{p})[w]$,

$C(f, \overset{A(\overline{p})}{\underset{e}{\bullet}}, w) = \underline{call}\ A(\overline{p})[w]$, if there is a leaf labelled $A(\overline{p})$ in e. This case gives rise to the binding $u(A(\overline{p})[w]) = C(f, e, w)$ in u,

$$C(f,\ \overset{A(\bar{p})}{\underset{e}{\bullet}},w) = C(f,e,w),\ \text{if no leaf in } e \text{ is labelled } A(\bar{p}),$$

$$C(f,\ \underset{e_1\quad e_2}{\bigwedge},w) = C(f,e_1,w);\ C(f,e_2,w),$$

$$C(f,\ \underset{e_1\quad e_2}{\overset{\pi}{\bigwedge}},w) = \pi \rightarrow C(f,e_1,w),\ C(f,e_2,w),\ \text{if } \pi \in \Pi.$$

It is easily seen, that for any expandable, simple language scheme $l \in L_s$ and program $p \in P$, $cg(F(l))(p)$ is in fact well-defined. This follows from the fact that each expansion tree generated contains only a finite number of variables ν, and that for all $p, F(l)(p)\downarrow 2$ is a subprogram of p.

Example 6: Consider the expansion tree for WhileDo of ex. 5. Let e be an expression and c a command, and let ι_e, ι_c denote the translations in I of e and c. Then the translation of a program WhileDo(e,c) is the program:

call WhileDo(e,c),

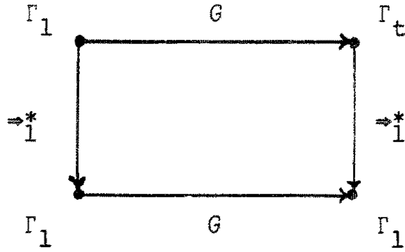
where the label WhileDo(e,c) is bound to:

ι_e;isTrue→ι_c; call WhileDo(e,c),I

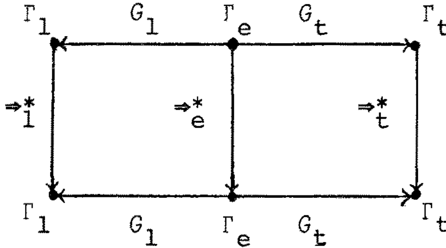
in the environment. □

8. The correctness proof

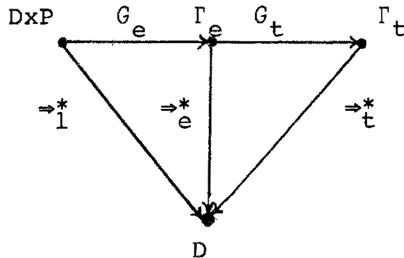
The correctness of cg - i.e. that for all expandable $l \in L_s$ and programs $p \in P$, $BT_l(l)(p) = BT_t(u)(cg(F(l))(p))$, where u is the environment generated by $cg(F(l))(p)$ - we should like to provide a generalization of cg: a function mapping configurations in T_l into configurations in T_t and then prove inductively that derivations in T_l and T_t corresponds under this mapping. Intuitively speaking, this is - in the terms of e.g. [Morris 73] - a proof that the relational diagram:



commutes - that is, that $\Rightarrow_1^* \circ G = G \circ \Rightarrow_t^*$, where G is the mapping mentioned above and $\circ$ denotes relational composition. However, G does not exist - except for configurations in Γ_1 which are in either DxP or D. Therefore we introduce an intermediate transition system T_e giving semantics for expansion trees, and prove instead commutativeness of the diagram:



Inasmuch as 1) G_1 is surjective, 2) G_1 and G_t acts as the identity on D, and 3) for configurations in Γ_1 of the form $<d,p>$ we can find an "expansion mapping" $G_e : \Gamma_1 \rightarrow \Gamma_e$ such that $d = G_1(G_e(<d,p>))\downarrow 1$ and $p = G_1(G_e(<d,p>))\downarrow 2$, commutativeness of this diagram implies commutativeness of:



which is clearly what we are aiming at.

9. Conclusion and directions for future work

We have been looking at relational semantics of programming languages, given in the structural operational style of [Plotkin 81]. We have presented and outlined a proof of the correctness of an algorithm for the generation of compilers from such semantics into a simple, somewhat structured target language, provided:

1) the language specified can be seen to be deterministic, and
2) the specification satisfies some technical constraints of expandability and disciplinedness of primitive operations.

The target language is easily linearized and extended to cover a wider range of control flow instructions (mainly conditional and unconditional jumps).

Like [Jones/Christiansen 81] the compiler generator works relative to an interpretation of data operations. In order to obtain a runnable system such an interpretation and an implementation of it in terms of a compiler should be provided-not a difficult task as in an essentially operational semantics states should be finite.

Much work remains to be done with respect to the practical applicability of the present approach. First, it should be tried out on a language of realistic complexity in order to assess the descriptive power of the metalanguage and the efficiency of the system. Secondly, efficiency should be improved.

We see mainly two sources of inefficiency. First, the control flow implementation should be optimized by detecting irreversible ("single-threaded", [Stoy 77]) state components. Some progress on this point have recently been made in the framework of denotational semantics ([Schmidt 85]).

Secondly, methods for distinguishing between compile time and run time should be investigated. This distinction could either be explicit as e.g. Plotkin's ([Plotkin 81]) proposal for specifying context-sensitive constraints or through the introduction of a dual type system as in [Nielson/Nielson 85],

or it could be implicit by somehow splitting states into compile- and run-time dependent parts - e.g. through some kind of data flow analysis. One might suspect that, due to the more rigid semantic metalanguage used (all states finite and all computations expressed in terms of transitions), problems in this direction might be more easily approached in the framework proposed here.

Acknowledgements

This work grew out of a thesis prepared for the degree of MSc at the Univeristy Centre of Aalborg, Denmark. Thanks are due to our advisor, Dr. F. Nielson, for many insightful comments and discussions during the preparation of our thesis.

References

[ADJ 78] J.A. Goguen, J.W. Thatcher, E.G. Wagner: *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* In: Current Trends in Programming Methodology, Vol. IV, R.T. Yeh (editor), Prentice-Hall 1978.

[Bjørner 77] D. Bjørner: *Formal Development of Interpreters and Compilers*, DtH ID673, 1977.

[De Bakker/ De Roever 72] J.W. De Bakker, W.P. De Roever: *A Calculus for Recursive Program Schemes.* In: Automata, Languages, Programming, Nivat (editor), North-Holland, Amsterdam 1972.

[Ganzinger 80] H. Ganzinger: *Transforming Denotational Semantics into practical Attribute Grammars.* In: Semantics-Directed Compiler Generation, LNCS 94, N.D. Jones (editor), 1980.

[Gordon 79] M.J.C. Gordon: *The Denotational Description of Programming Languages - An Introduction*, Springer-Verlag 1979.

[Greif/Meyer 81] I.Greif, A.R. Meyer: *Specifying the Semantics of while Programs: A Tutorial and Critique of a Paper by Hoare and Lauer*, ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, Oct. 1981.

[Hoare/Lauer 73] C.A.R. Hoare, P.E. Lauer: *Consistent and Complementary Formal Theories of the Semantics of Programming Languages*, Acta Informatica 3, 1974.

[Jensen/Dam 85] F. Jensen, M. Dam: *Automatisk generering af oversættere udfra operationelt semantiske definitioner af programmeringssprog,* M.Sc. Thesis (in danish), Aalborg University Centre, 1985.

[Jones/Christiansen 81] N.D. Jones, H. Christiansen: *Control Flow Treatment in a Simple Semantics-Directed Compiler Generator,* DAIMI PB-137, sept. 1981.

[Jones/Schmidt 80] N.D. Jones, D.A. Schmidt: *Compiler Generation from Denotational Semantics.* In: Semantics-Directed Compiler Generation, LNCS 94, N.D. Jones (editor), 1980.

[Kowalski 74] R. Kowalski: *Predicate Logic as Programming Language.* In: Information Processing 74, North-Holland 1974.

[Milne/Strachey 76] R. Milne, C. Strachey: *A Theory of Programming Language Semantics,* Chapman and Hall, 1976.

[Morris 73] F.L. Morris: *Advice on Structuring Compilers and Proving Them Correct,* Proc. 2nd ACM Symp. on Principles of Prog. Lan., 1973.

[Mosses 79] P. Mosses: *SIS-Semantics Implementation System,* Reference Manual and User's Guide, DAIMI MD-30, 1979.

[Mosses 80] P. Mosses: *A Constructive Approach to Compiler Correctness.* In: Semantics-Directed Compiler Generation, LNCS 94, N.D. Jones (editor), 1980.

[Nielson 84] H.R. Nielson: *Hoare Logic's for Run-time Analysis of Programs,* Ph.D. Thesis, University of Edinburgh, oct. 1984.

[Nielson/ Nielson 85] F. Nielson, H.R. Nielson: *Pragmatic Aspects of Two-Level Denotational Meta-Languages*, AUC R-85-13.

[Plotkin 81] G.D. Plotkin: *A Structural Approach to Operational Semantics*, DAIMI FN-19, sept. 1981.

[Schmidt 85] D.A. Schmidt: *Detecting Global Variables in Denotational Specifications*, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 2, april 1985.

[Sethi 83] R. Sethi: *Control-Flow Aspects of Semantics-Directed Compiling*, ACM Transactions on Programming Languages and Systems, Vol. 5, No. 4, oct. 1983.

[Stoy 77] J.E. Stoy: *Denotational Semantics:* The Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977.

[Wand 82] M. Wand: *Deriving Target Code as a Representation of Continuation Semantics*, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, july 1982.