Formal Aspects of Computing (2000) 12: 71–99

Formal Aspects of Computing

Natural Semantics-Directed Generation of Compilers and Abstract Machines

Stephan Diehl

Universität des Saarlandes, Saarbrücken, Germany

Abstract. In this paper we present the motivation, theory and transformations of our semantics-directed compiler generator. The main novelty of our generator is that it generates compilers and abstract machines. The execution times of the abstract machine programs produced by our generated compiler compare well to those of target programs produced by compilers generated by other semantics-directed generators. The generated specifications of compilers and abstract machines are suitable as a starting point for handwriting compilers and abstract machines. Our generator is fully automated and its core transformations are proved correct.

Keywords: Abstract machines; Natural semantics; Compiler generation

1. Introduction

In traditional compiler design the work of a compiler is divided into several phases: lexical, syntactical and semantical analysis, optimisations and code generation. For several of these phases generators exist – most prominently LEX and YACC for generating lexical and syntactical analysers. A common feature of all generators is that the phase in the compiler is described using a meta-language (e.g., regular expressions or context-free grammars) and that the generator produces the related compiler module. There exist several good textbooks on compiler design[ASU86, WiM92, Lem92]. However, all of these books present ready-made mappings from source language constructs to target language constructs, the so-called translation schemes, instead of deriving them. Hence, the reader is expected to learn how to design code generators by analysing translation schemes as opposed to from first principles. The same is true for abstract machines. Abstract machines are virtual target architectures which support the concepts of the source language. Typically abstract machines are presented together with translation schemes from the source language to the abstract machine language. There is only little work on how translation schemes and abstract machines are designed.

The aim of our work [Die96] is to detect underlying principles that relate abstract machines to programming language semantics, and to automate part of the design process for abstract machines. Thus, we need to ensure that the behaviour of a source program will be maintained by translating it into the abstract machine language, and then applying the abstract machine.

Correspondence and offprint requests to: S. Diehl, FB-14 Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany. Email: diehl@cs.uni-sb.de

The behaviour of a program will depend on its semantics. Often this aspect of a programming language is only described in natural language which is both ambiguous and vague. We shall use formal techniques to describe the meaning of programs in a particular language and to prove that our transformations are correct.

1.1. Outline

In this article we will provide an answer to the question, how can one generate translation schemes and abstract machines. It is based on pass separating a natural semantics specification. One requirement of a compiler is, that it is complete in the sense that every correct program can be compiled. The second approach is not fully automatic and does not guarantee completeness. The approach presented in this article both guarantees completeness and is fully automatic. As far as we know, our generator is the first running implementation of a system which automatically produces both compilers and abstract machines from a semantics specification. This article contains

syntax and semantics of our semantics notation: We present the notation called Two-Level Big-Step Semantics. Specifications written in this notation can be processed by our system.

a presentation of our generator for abstract machines: From a natural semantics specification the generator automatically produces a compiler and abstract machine. Whereas all existing semantics-directed compiler generators use partial evaluation or a direct translation into a fixed target language, we chose pass separation as the key transformation of our system.

an experimental evaluation of the generator: We tested the generator with semantics specifications of two toy languages (SIMP and Mini-ML) and a specification of Action Notation. For Mini-ML we got an abstract machine which is very close to the CAM, an abstract machine used for efficient implementations of ML.

a sketch of the correctness proof of the generator: We give the main correctness results and a sketch of the full correctness proof contained in [Die96].

In our work, composing source-to-source transformations plays a central role. This divide-and-conquer approach to compiler generation has some advantages over a more direct approach. Each transformation introduces a new property; it transforms one kind of representation into a more restricted kind. As a consequence the transformations can be developed, debugged, and replaced separately and to a certain extent each transformation can be understood and proved correct in isolation. Composing several transformations leads to a modular structure of our system. This modular structure facilitates extending the system over time to include more powerful analyses and transformation methods.

Abstract machines provide intermediate target languages for compilation. First, the compiler generates code for the abstract machine. Then, this code can be interpreted or further compiled into real machine code. Abstract machines as an intermediate stage increase portability and maintainability of compilers. The instructions of an abstract machine are tailored to specific operations required to implement operations of a source language. The structure of architectures called abstract machines varies widely. There are three levels at which one might define an abstract machine: the way it is specified (see abstract interpreter in Section 5.2.10), the way it works (executes instructions and changes the state), and the purpose it is used for (intermediate language for compilation).

For almost all kinds of languages, there exist abstract machines [DHS99], e.g.:

Type of Language	Abstract Machines
imperative	P4 [PeD82]
functional	SECD [Lan64], CAM [CCM85], FAM [Car84], G-Machine [Joh84]
logic	WAM [War77, Aït91]
functional/logic	CAMEL [Müc92]
constraint	CLAM [JSM92]
concurrent, constraint	OZAM [MSS95]
object oriented	[BFH94], JVM [Sun95]*

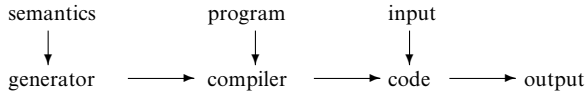
* In JAVA, which is a widely used application for the World Wide Web (WWW), sending abstract machine code over the network helps to cope with two new problems: the heterogeneity of the network and security (virus protection).

Abstract machines are usually designed in an ad-hoc manner often based on experience with other abstract machines. But also some systematic approaches have been investigated. One of those is based on partial evaluation of example programs [Kur86, Nil93, Die97a]. Another approach is to use pass separation transformations [JøS86]. John Hannan [Han94] introduced a pass separation transformation, which splits a set of term rewriting rules representing an abstract interpreter into two sets of term rewriting rules: the first set represents a compiler in an abstract machine language, while the second set represents an abstract machine. Since rewrite rules are a poor language to specify interpreters, Stephen McKeever [McK94] extended Hannan's transformations to determine inductive rules. In McKeever's framework, the factorisation algorithm of Fabio daSilva [daS90] plays a central role. By hand, McKeever transformed a natural semantics specification for an imperative toy language with while-loops into a compiler. We formally defined similar transformations and implemented them in Prolog. Applying our system to the above mentioned toy language yields similar results.

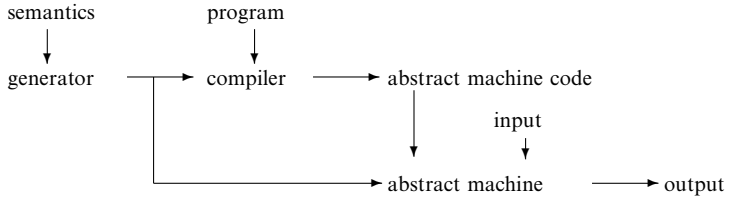
The contributions of our work are

- the formal definition of a meta-language and the transformations;
- the implementation of the method;
- its application to the specifications of realistic programming languages;
- and thus the detection of missing links (e.g., extension of factorisation to more than two rules) and insufficiencies in previous work;
- the development of optimization transformations; and
- the correctness proof of the core transformations.

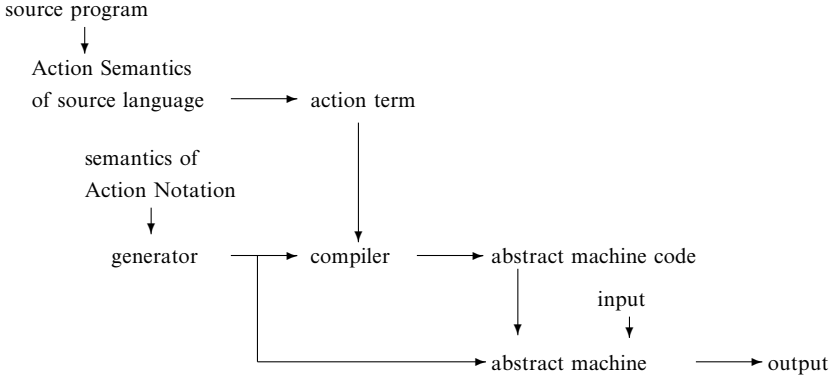
Given a semantics specification of a source language, current semantics-directed compiler generators produce compilers from the source language into a fixed target language.



Rather than just generating compilers which translate source programs into a fixed target language, our system generates both a compiler and an abstract machine. The generated compiler translates source programs into code for the abstract machine.



We chose Action Notation [Mos92] as an example of a realistic programming language, because it offers a rich set of primitives underlying both imperative and functional programming languages. Since Action Notation is used to write Action Semantics specifications, we can then combine the generated compiler for Action Notation with an Action Semantics specification of a programming language. As a result, we get a compiler from the programming language to the generated abstract machine language for Action Notation.



Before going on to the transformations of our generator, we have to define our meta-language and its semantics.

2. Two-Level Big-Step Semantics (2BIG)

Natural semantics has been used by programming language researchers to specify many aspects of programming languages, e.g., program analyses [Sch95], type systems [DaM82], translations and optimisations [Mou93], static semantics and dynamic semantics [MTH90]. Example specifications of type systems, dynamic semantics and translation to abstract machine code for Mini-ML are given in [Kah87, Des84, Des86]. In this article we use natural semantics to specify the dynamic semantics of programming languages. Next we define our notation to write natural semantics specifications and give an example specification.

2.1. Natural Semantics

After reading several papers related to natural semantics we believe that there is some confusion of what natural semantics [Kah87] is and how it differs from structural operational semantics (SOS) [Plo81]. In summary we found the following partly contradictory views:

1. Natural semantics is a certain style of SOS [Mos92, Win93].
2. In SOS we specify individual state transitions, whereas in natural semantics overall results of evaluations are specified [NiN92]. This difference is also emphasised by the terms small-step and big-step semantics [Win93] or transitional style and relational style semantics [daS92]. At least in [NiN92] the authors seem to be unaware of the fact, that although preferring the transitional style Plotkin uses both styles in his landmark notes [Plo81, page 40].
3. In SOS an inductive system is defined by inductive rules [Ast91]. Properties of the language defined can be proved by rule induction (see Section 2.3.1). Natural semantics rules are most reminiscent of natural deduction rules; more precisely sequent calculus and the semantics rules are treated as proof rules [Kah87, Des86, Pet95]. In natural deduction assumptions can be arbitrary formulas, whereas in natural semantics we usually have assumptions on variables of the language being defined. 'Furthermore, semantics written in this style appears rather intuitive, so that **natural** may also be understood in the lay-man's sense.' [Des86].
4. Natural semantics is similar to relational style SOS, but its rules can't necessarily be given an operational reading [Ber91, Des86]. According to this view of natural semantics cyclic dependencies are possible as for E^* in $\frac{P_2 \triangleright [[X \mapsto E^*]|E] \rightarrow E^* \quad P_1 \triangleright [[X \mapsto E^*]|E] \rightarrow E'}{\textbf{letrec } X = P_2 \textbf{ in } P_1 \textbf{ end} \triangleright E \rightarrow E'}$. The operational reading would say that P is executed in state $[[X \mapsto E^*]|E]$, but E^* is not known until P is executed.

The main source of this confusion is that neither Plotkin nor Kahn precisely defined their meta-languages but directly dived into examples:

> The above rules are easily turned into a formal system of formulae, axioms and rules. ... However, the present work is too explorating for us to fix our ideas, although we may later try out one or two possibilities. [Plo81, page 33]

As we had to implement our meta-language and prove the correctness of transformations, it was absolutely necessary to define the semantics of the meta-language precisely. We call our meta-language 2BIG. It is a certain style of SOS, namely big-steps semantics. With the exception of not allowing for cyclic dependencies, it can also be regarded as a notation to write natural semantics specifications.

2.2. Syntax

We discuss the syntax of 2BIG, a language designed to write natural semantics specifications. The language combines the structural approach of natural semantics [Kah87] with the separation of general and implementation details. This is achieved by providing an interpretation for function symbols in the rules in a separate specification. We refer to the specification of the interpretation for function symbols as the second level and the inference rules as the first level. All transformations are syntactical transformations on the inference rules; the interpretations of functions remain unchanged. All compile-time computations have to be in the first

x		variable symbol	
c		constructor symbol	
f		function name	
p		predicate name	
$\mathbf{T}$	::=	$f(\mathbf{T}^*) \mid \tilde{\mathbf{T}}$	terms with functions
$\tilde{\mathbf{T}}$	::=	$c(\tilde{\mathbf{T}}^*) \mid x$	terms without functions
$\mathbf{S}$	::=	$c(\mathbf{T}^*) \triangleright \mathbf{T} \rightarrow \tilde{\mathbf{T}}$	transitions with functions
$\tilde{\mathbf{S}}$	::=	$c(\tilde{\mathbf{T}}^*) \triangleright \tilde{\mathbf{T}} \rightarrow \mathbf{T}$	transitions without functions
$\mathbf{Q}$	::=	$p(\mathbf{T}^*) \mid \mathit{not}\ p(\mathbf{T}^*)$	side conditions
$\mathbf{J}$	::=	$\mathbf{S} \mid \mathbf{Q}$	judgements
$\mathbf{R}$	::=	$\frac{\mathbf{J}^*}{\mathbf{S}}$	rules

Although we define terms here in prefix notation, we will use postfix or 'mixfix' notation, if it is more convenient, e.g., **while** B **do** C **od** instead of **while**(B, C).

Fig. 1. Syntax of 2BIG.

level; run-time aspects can be hidden in the second level. Only the names and signatures of the function symbols are made available to the first level. As we do not change the interpretation of functions, we specify functions only informally here. In our system the user has to provide Prolog, SML or C implementations of these functions.

We do not use the term Two-Level in the sense of [NiN86, JGS93], where programs can be annotated to distinguish compile-time and run-time expressions, but as in Peter Lee's High-Level semantics [Lee89]. In High-Level semantics the first level consists of semantics equations similar to those of denotational semantics and is called macrosemantics. In the right hand sides of these equations operators are employed. The interpretation of these operators is specified separately and is called microsemantics.

In 2BIG the dynamic semantics of a programming language is defined by a set of inference rules. The syntax of such 2BIG rules is given in Fig. 1.

The following is an example of a 2BIG rule: $r = \frac{member((X \mapsto Y), S) \qquad V \triangleright S \rightarrow N}{assign(X,V) \triangleright S \rightarrow replace(X,S,N)}$

We will use the notational convention that meta-variables $c, c', c_i, e, e', e_i, \ldots$ denote terms. It will be helpful to introduce some terminology about rules and their components. **Judgements** are transitions of the form $c \triangleright e_1 \rightarrow e_2$, e.g., $assign(X, V) \triangleright S \rightarrow replace(X, S, N)$, or side conditions of the form $p(t_1, \ldots, t_n)$ or $not\ p(t_1, \ldots, t_n)$, e.g., $member((X \mapsto Y), S)$.

We will use the term **left hand side** (LHS) to refer to $c \triangleright e_1$ in a transition and the term **right hand side** (RHS) to refer to e_2. In a rule the judgements above the line are called **preconditions** and the judgement below the line is called the **conclusion**. The variables in a term t are denoted by $\mathcal{V}(t)$ and those variables which only occur once in a rule are called anonymous. In the above example, $\mathcal{V}(r) = \{X, V, S, N, Y\}$ holds and Y is an **anonymous variable**. Furthermore we adopt the notation for list constructors from Prolog, e.g., $[1, 2, 3] = [1|[2|[3|[]]]]$. We will give a meaning to the 2BIG language after the transformation of side conditions, i.e., we regard side conditions as syntactic sugar, which can be transformed into transitions containing the characteristic functions of the predicates as described in Section 5.2.2.

Functions (e.g., *replace* and the characteristic function of *member*) are defined separately, e.g., as Prolog predicates. We refer to their definitions as the second level of the specification. All transformations presented here only manipulate the first level, i.e., the inference rules.

In a transition $c \triangleright e \rightarrow e'$ the meta-variables c, e and e' denote terms, thus they are not different entities. But to emphasize, that c, e and e' occur at different positions in a transition, we call the term on the left of $\triangleright$ the **instruction**, the terms on the right of $\triangleright$ and $\rightarrow$ are called **states** and we will refer to the outermost constructor of an instruction as an **instruction symbol**. This convention is motivated by the way transitions are used in semantics specifications. Usually a transition of the form $c \triangleright e_1 \rightarrow e_2$ is interpreted as 'the execution of the instructions c in state e_1 yields state e_2'. By instructions we mean the constructs of the language being defined and by state we mean run-time information like bindings, environments or stores. Some authors refer to instruction–state pairs as configurations. As a deviation from most work in natural semantics we use $c \triangleright e$ instead of $e \vdash c$. As the rules usually define different cases for c, not for e, we feel that our notation is more readable.

2.3. Rule Induction

Before we can define the semantics of 2BIG , we have to define some mathematical preliminaries. The following definitions are based on those given in [Acz77, daS92, daS90].

2.3.1. Inductive Systems

Definition 2.1 (Inductive System). Let U be a set. An **inductive rule** is a pair (P, c) where $P \subseteq U$ and $c \in U$. We call P the premises and c the conclusion. An **inductive system** ϕ is a set of inductive rules. ϕ defines a subset of U.

Definition 2.2 (ϕ-Closed). $A \subseteq U$ is **ϕ-closed** iff for all $(P, c) \in \phi$ we have that $P \subseteq A$ implies $c \in A$.

Definition 2.3 (Inductively Defined Set). The **set inductively defined by** ϕ is defined as $\mathscr{I}(\phi) = \bigcap\{A | A \text{ is } \phi - closed\}$.

For example the set of all conclusions $\{c|(P, c) \in \phi\}$ is ϕ-closed.

Theorem 2.1. $\mathscr{I}(\phi)$ is ϕ-closed

Proof. $P \subseteq \mathscr{I}(\phi) \overset{2.3}{\Longrightarrow}$ for every ϕ-closed set A: $P \subseteq A \overset{2.2}{\Longrightarrow}$ for every ϕ-closed set A: $c \in A \overset{2.3}{\Longrightarrow} c \in \mathscr{I}(\phi)$. □

Since $\mathscr{I}(\phi)$ is the intersection of all ϕ-closed sets, we have that $\mathscr{I}(\phi)$ is the least ϕ-closed set.

The infinite, inductive system $\phi_1 = \{(\{m, n\}, p) | n, m \in \mathscr{N}, p = m * n\} \cup \{(\{\}, 2)\}$ defines the set $\mathscr{I}(\phi_1) = \{2^n | n \in \mathscr{N}, n > 0\}$.

Theorem 2.2 (Principle of Rule Induction). Let Ψ be a predicate over U. $(\forall (P, c) \in \phi : (\forall x \in P : \Psi(x) \text{ is true}) \Rightarrow \Psi(c) \text{ is true}) \Rightarrow (\forall a \in \mathscr{I}(\phi) : \Psi(a) \text{ is true})$

Next we formally define a proof based on inductive rules as a sequence of items. Each item in the sequence is either an axiom or it follows by a subset of items preceding that item in the sequence. An inductive system is **finitary** if the preconditions of all rules are finite.

Definition 2.4 (Finite Length Proof). Let ϕ be a finitary inductive system and $b_0, \ldots, b_n, b \in U$. A sequence $\langle b_0, \ldots, b_n \rangle$ is a finite ϕ-proof of b if $b_n = b$ and for all $m \leqslant n$ there is a set $B \subseteq \{b_i : i < m\}$ such that $(B, b_m) \in \phi$.

Theorem 2.3 (proof in [daS92]). For every finitary inductive system ϕ we have that $\mathscr{I}(\phi) = \{b : b \text{ has a finite } \phi\text{-proof}\}$.

$8 \in \mathscr{I}(\phi_1)$ because $\langle 2, 4, 8 \rangle$ is the shortest ϕ_1-proof of 8.

In a ϕ-proof it is not obvious for an item what subset of the sequence implied it. The structure of the proof is made explicit by ϕ-trees.

Definition 2.5 (Proof Tree). Given a finitary inductive system ϕ, a ϕ-tree (or proof tree) of b, denoted $PT_\phi(b)$, is an object $\frac{PT_\phi(b_1) \ \ldots \ PT_\phi(b_n)}{b}$ where there exists a rule $(\{b_1, \ldots, b_n\}, b) \in \phi$, such that $PT_\phi(b_i)$ is a ϕ-tree for b_i and $(1 \leqslant i \leqslant n)$.

Theorem 2.4 (proof in [daS92]). For every finitary inductive system ϕ we have that $\mathscr{I}(\phi) = \{b : b \text{ has a finite } \phi\text{-tree}\}$.

$8 \in \mathscr{I}(\phi_1)$ because $\frac{\frac{\bar{2} \quad \bar{2}}{4} \quad \bar{2}}{8}$ and $\frac{\bar{2} \quad \frac{\bar{2} \quad \bar{2}}{4}}{8}$ are ϕ_1-trees.

Inductively defined sets can also be seen as least fixed-points of monotonic transformations:

Theorem 2.5 (proof in [Ast91]). Let $\Psi(X) = \{c|(P, c) \in \phi, P \subseteq X\}$.
Then $\mathscr{I}(\phi)$ is the least fixed-point of Ψ.

2.3.2. Relational Inductive Systems

Let $T_\Sigma(X)$ be the set of all first-order terms over a signature Σ with variables in X. Terms without variables are ground. The set of all **ground terms** over a signature[1] Σ is denoted by T_Σ. Let $t \in T_\Sigma(X)$ and θ be a substitution such that $\theta(t) \in T_\Sigma$, then $\theta(t)$ is a ground instance of t.

Definition 2.6 (Relational Inductive System). A **relational inductive rule** is a pair (P, c), where P is a finite subset of $T_\Sigma(X)$, and $c \in T_\Sigma(X)$. A set of relational inductive rules is called a relational inductive system.

Using variables, we can now define a finite, relational inductive system, which defines the same set as the infinite, inductive system in the preceding examples. We use capitals for variables: $\phi_2 = \{(\{M, N\}, mult(M, N)), (\{\}, 2)\}$

Definition 2.7 (Evaluation of Functions). Assume there is a division of the names in Σ into function names F and constructor names C, such that $\Sigma = F \cup C$ and $F \cap C = \emptyset$. Furthermore let $\alpha : F \to N$ map each function name to its arity, $m = max(\{\alpha(f) : f \in F\})$ and let $\sigma : F \to ((T_C{}^0 \cup \ldots \cup T_C{}^m) \to (T_C \cup \{\bot\}))$ be an interpretation of the function names where $\bot$ denotes non-termination of a computation.

The evaluation $\eta_\sigma(t)$ of a ground term t by an interpretation σ is defined as:
Let $t = f(t_1, \ldots, t_n)$,

1. $f \in F$: if $\forall i : \eta_\sigma(t_i) \neq \bot$ then $\eta_\sigma(t) = \sigma(f)(\eta_\sigma(t_1), \ldots, \eta_\sigma(t_n))$, else $\eta_\sigma(t) = \bot$.
2. $f \in C$: if $\forall i : \eta_\sigma(t_i) \neq \bot$ then $\eta_\sigma(t) = f(\eta_\sigma(t_1), \ldots, \eta_\sigma(t_n))$, else $\eta_\sigma(t) = \bot$.

η_σ naturally extends to sets and tuples, e.g., let S be a set, then $\eta_\sigma(S) = \{\eta_\sigma(x) : x \in S\}$.

The signature in the above example is $\Sigma = \mathcal{N} \cup F$ where $F = \{mult\}$ and the interpretation σ maps *mult* onto the multiplication of natural numbers.

Definition 2.8 (Derived Inductive System). The **inductive system $\overline{\phi}$ derived from a relational inductive system** ϕ by an interpretation σ is the set of all rules $\eta_\sigma(\theta((\{p_1, \ldots, p_n\}, c)))$, such that $(\{p_1, \ldots, p_n\}, c) \in \phi$, and there exists a substitution θ such that $\theta(p_1), \ldots, \theta(p_n), \theta(c)$ are ground instances.

The derived inductive system for our example is $\overline{\phi_2} = \{(\{m, n\}, p) | m, n \in \mathcal{N}, p = m * n\} \cup \{(\{\}, 2)\}$ and this is equal to the inductive system ϕ_1 of the preceding example. As a consequence we have that the set defined by our relational inductive system is the set defined by the derived inductive system: $\mathcal{I}(\overline{\phi_2}) = \mathcal{I}(\phi_1) = \{2^n | n \in \mathcal{N}, n > 0\}$.

2.4. Semantics of 2BIG

After transformation of side conditions into transitions (see Section 5.2.2), the 2BIG rules are simply relational inductive rules with ordered premises where we regard $\to$ as a ternary constructor symbol. Let ϕ be the set of these relational inductive rules and $\overline{\phi}$ be the derived inductive rule set. The semantics of a program c in a state e is the set of all states e', such that $c \triangleright e \to e' \in \mathcal{I}(\overline{\phi})$, which means that there is a $\overline{\phi}$-proof of $c \triangleright e \to e'$. Note, that c, e and e' are first order terms, which we interpret as programs and states.

2BIG rules can also be given a procedural reading as in logic programming [Llo87]. Informally, to prove that a transition $c \triangleright e_1 \to e_2$ (goal) follows from the inference rules, we unify it with the conclusion of a rule. If it unifies then the preconditions of that rule become our new goals. If the rule has no preconditions, then the goal trivially follows from that rule. This procedural reading underlies the Prolog implementation of 2BIG.

2.5. Properties of 2BIG Rules

In the sequel we assume that the preconditions of 2BIG rules are ordered sets, then we write $\frac{B}{c \triangleright e \to e'}$ as the pair $(B, c \triangleright e \to e')$.

[1] For simplicity we do not consider many-sorted signatures as in [daS92] here. Admittedly they are more adequate, since one can use the same function name for different purposes in a many-sorted signature, i.e., they allow operator overloading.

Definition 2.9 (Deterministic Rules). A set ϕ of 2BIG rules is **deterministic**, iff for all rules $(B_1, c_1 \triangleright e_1 \rightarrow e_1'), (B_2, c_2 \triangleright e_2 \rightarrow e_2') \in \phi$ holds that the terms c_1 and c_2 and the terms e_1 and e_2 are not unifiable, i.e. there is no substitution θ such that $(c_1, e_1)\theta = (c_2, e_2)\theta$.

Next we define a sufficient property of 2BIG rules. Rules which have this property can be transformed into deterministic rules. By $=_\alpha$ we will denote equality of terms and formulae modulo renaming of variables.

Definition 2.10 (Determinate Rules). A set ϕ of 2BIG rules is **determinate**, iff for all pairs of rules $(B_1, c_1 \triangleright e_1 \rightarrow e_1'), (B_2, c_2 \triangleright e_2 \rightarrow e_2') \in \phi$ we have that

1. c_1, e_1 and c_2, e_2 are not unifiable or
2. $c_1, e_1 =_\alpha c_2, e_2$ and $B_1 = \{c_{11} \triangleright e_{11} \rightarrow e_{11}', \ldots, c_{1m_1} \triangleright e_{1m_1} \rightarrow e_{1m_1}'\}$, $B_2 = \{c_{21} \triangleright e_{21} \rightarrow e_{21}', \ldots, c_{2m_2} \triangleright e_{1m_2} \rightarrow e_{2m_2}'\}$ and there exists a renaming θ of variables and an index j, such that $(c_{1j}, e_{1j})\theta = (c_{2j}, e_{2j})\theta$, e_{1j}' and e_{2j}' are not unifiable, and $\forall k < j : (c_{1k}, e_{1k}, e_{1k}')\theta = (c_{2k}, e_{2k}, e_{2k}')\theta$.

We call the smallest index j', such that $c_{1j'}, e_{1j'} \neq_\alpha c_{2j'}, e_{2j'}$ the **discrimination index**.

Definition 2.11 (Defining Occurrence). An occurrence of a variable in a rule is **defining**, iff it is on the left hand side of the conclusion or if there is no defining occurrence of the variable in the conclusion, but the variable occurs on the right hand side of a precondition and there is no occurrence of the variable in a precondition prior to this one. All other occurrences are **using**.

Definition 2.12 (Well-Orderedness). A rule is **well-ordered**, iff every using occurrence of a variable is preceded by a defining occurrence of that variable.

Definition 2.13 (Sequential Rules). A set ϕ of 2BIG rules is **sequential**, iff for all rules $(\{c_1 \triangleright e_1 \rightarrow e_1', \ldots, c_m \triangleright e_m \rightarrow e_m'\}, c \triangleright e \rightarrow e') \in \phi$ we have that $e_i' = e_{i+1}$ (for all $1 \leqslant i < m$) and $e_m' = e'$.

3. Static Semantics

In the rest of this paper we assume that the 2BIG rules are determinate, well-ordered and linear and have ordered preconditions.

Ordered Preconditions. In 2BIG specifications there is a fixed order of preconditions. When we model 2BIG specifications by relational, inductive rules we do not need this restriction and write the list of preconditions as a set of premises.

Determinacy. Specifications in 2BIG have to be determinate [daS90]. Consequently, whenever two rules have conclusions with unifiable left hand sides, at most one of the rules can be successfully applied to prove a goal. The restriction to determinate rule sets is important, because determinate rule sets can be converted into deterministic ones, i.e., at most one rule will have a conclusion, which unifies with a goal. Deterministic rules can be converted into term rewriting rules and finally these rewrite rules can be pass separated into rewrite rules for a compiler and an abstract machine. In the next section, these transformations are discussed in more detail.

Well-Orderedness. Furthermore the 2BIG rules must be well-ordered, i.e., every variable must be defined before it can be used. This property does not allow for cyclic dependencies of variables and enables us to use term rewriting and not graph rewriting systems for the generated abstract machines.

Linearity. As another restriction of the rule syntax we have that a variable must not occur twice on the left hand side of the conclusion. We need this property to ensure that the generated term rewriting systems are linear.

4. Example: A 2BIG Specification

We consider a small fragment of a 2BIG specification of SIMP, an imperative toy language used to teach formal semantics in [Win93]. The fragment suffices to construct a proof tree for the program **while i>1 do i:=i−1 od**.

First we specify how expressions are evaluated.[2] In this example the state will be a mapping of variable names to values. A transition of the form $E \triangleright S \to V$ means that in state S the expression E evaluates to V.

$$\frac{is\_id(X)}{X \triangleright S \to lookup(X,S)} \qquad \frac{is\_num(N)}{N \triangleright S \to N} \qquad \frac{E_1 \triangleright S \to V_1 \quad E_2 \triangleright S \to V_2}{E_1 > E_2 \triangleright S \to greater(V_1,V_2)} \qquad \frac{E_1 \triangleright S \to V_1 \quad E_2 \triangleright S \to V_2}{E_1 - E_2 \triangleright S \to sub(V_1,V_2)}$$

Next the execution of assignments, sequencing and the *while*-loop are specified. Here a transition $A \triangleright S \to S'$ means that the execution of the statement A in state S yields state S'.

$$\frac{E \triangleright S \to V}{X := E \triangleright S \to replace(X,S,V)} \qquad \frac{C_1 \triangleright S \to S' \quad C_2 \triangleright S' \to S''}{C_1; C_2 \triangleright S \to S''}$$

$$\frac{B \triangleright S \to false}{\textbf{while } B \textbf{ do } C \textbf{ od} \triangleright S \to S} \qquad \frac{B \triangleright S \to true \quad C; \textbf{while } B \textbf{ do } C \textbf{ od} \triangleright S \to S'}{\textbf{while } B \textbf{ do } C \textbf{ od} \triangleright S \to S'}$$

Here states are bindings. A binding is a list of associations $x \mapsto y$, i.e., the key x is associated with the value y. In these rules the following functions have been used: $lookup(X,S)$ yields the value associated with the identifier bound to $0X$ in the binding S; $greater(V_1,V_2)$ yields *true* if the value V_1 is greater than the value V_2; $sub(V_1,V_2)$ yields the difference of the value V_1 and the value V_2; and $replace(X,S,V)$ yields a new binding, which differs from S only in that the association of the identifier X is replaced by an association of the identifier X to the value V.

And the side conditions in the example are: $is\_id(X)$ is *true* if X is an identifier and $is\_num(N)$ is *true* if N is a number.

Using the above 2BIG rules we can now construct a proof tree[3] for the program

while i>1 do i:=i−1 od

in the state $[\mathbf{i} \mapsto 2]$.

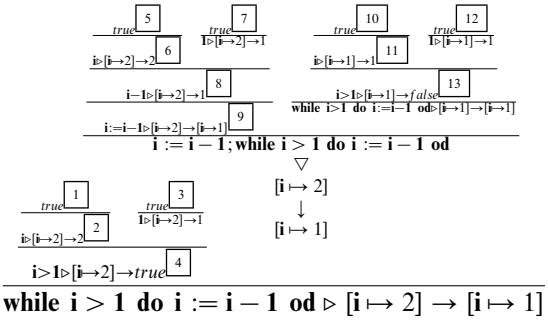
First we apply the second rule for **while**:

$$\frac{\mathbf{i} > \mathbf{1} \triangleright [\mathbf{i} \mapsto 2] \to true \quad \mathbf{i} := \mathbf{i} - \mathbf{1}; \textbf{while } \mathbf{i} > \mathbf{1} \textbf{ do } \mathbf{i} := \mathbf{i} - \mathbf{1} \textbf{ od} \triangleright [\mathbf{i} \mapsto 2] \to S'}{\textbf{while } \mathbf{i} > \mathbf{1} \textbf{ do } \mathbf{i} := \mathbf{i} - \mathbf{1} \textbf{ od} \triangleright [\mathbf{i} \mapsto 2] \to S'}$$

To prove the premises of the rule we apply the rules for the $>$ and **;** operators:

$$\frac{\dfrac{\mathbf{i} \triangleright [\mathbf{i} \mapsto 2] \to V_1 \quad \mathbf{1} \triangleright [\mathbf{i} \mapsto V_2]}{\mathbf{i} > \mathbf{1} \triangleright [\mathbf{i} \mapsto 2] \to true \equiv greater(V_1,V_2)} \quad \dfrac{\mathbf{i} := \mathbf{i} - \mathbf{1} \triangleright [\mathbf{i} \mapsto 2] \to S^* \quad \textbf{while } \mathbf{i} > \mathbf{1} \textbf{ do } \mathbf{i} := \mathbf{i} - \mathbf{1} \textbf{ od} \triangleright S^* \to S'}{\mathbf{i} := \mathbf{i} - \mathbf{1}; \textbf{while } \mathbf{i} > \mathbf{1} \textbf{ do } \mathbf{i} := \mathbf{i} - \mathbf{1} \textbf{ od} \triangleright [\mathbf{i} \mapsto 2] \to S'}}{\textbf{while } \mathbf{i} > \mathbf{1} \textbf{ do } \mathbf{i} := \mathbf{i} - \mathbf{1} \textbf{ od} \triangleright [\mathbf{i} \mapsto 2] \to S'}$$

If we continue this construction we finally get the following proof tree:



In the above proof, the results of evaluating functions and side conditions have been marked by boxed

[2] Some authors prefer to distinguish syntactic representation and semantic values by using functions from syntactic values to semantic values, e.g., *valuation* or *token_of*:

$$\frac{is\_num(N)}{N \triangleright S \to valuation(N)} \qquad \frac{is\_id(X)}{X \triangleright S \to lookup(token\_of(X),S)}$$

[3] We formally defined proof trees in Section 2.3.1.

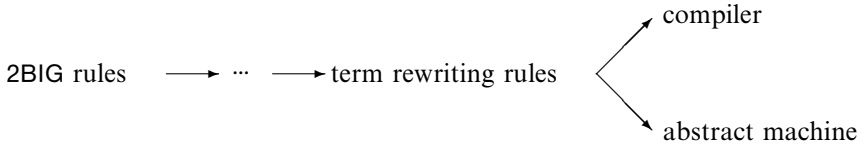
numbers $\boxed{n}$. Below we list these functions and side conditions.

$\boxed{1}$ $is\_id(\mathbf{i})$ $\boxed{2}$ $lookup(\mathbf{i}, [\mathbf{i} \mapsto 2])$ $\boxed{3}$ $is\_num(1)$ $\boxed{4}$ $greater(2,1)$
$\boxed{5}$ $is\_id(\mathbf{i})$ $\boxed{6}$ $lookup(\mathbf{i}, [\mathbf{i} \mapsto 2])$ $\boxed{7}$ $is\_num(1)$ $\boxed{8}$ $sub(2,1)$
$\boxed{9}$ $replace(\mathbf{i}, [\mathbf{i} \mapsto 2], 1)$ $\boxed{10}$ $is\_id(\mathbf{i})$ $\boxed{11}$ $lookup(\mathbf{i}, [\mathbf{i} \mapsto 1])$ $\boxed{12}$ $is\_num(1)$
$\boxed{13}$ $greater(1,1)$

5. Generation of Compilers and Abstract Machines

5.1. A Motivating Example

The system which we are going to present converts 2BIG rules into term rewriting rules. Then it generates from these term rewriting rules two sets of term rewriting rules: a set which defines a compiler and a set which defines an abstract machine.



In this section we look at the transformations of two simple 2BIG rules. It is impossible to give the motivation for the design of a single transformation without considering our intermediate goal to generate term rewriting rules. Before each transformation we discuss a problem which keeps us from converting the rules into term rewriting rules. Then we show the transformed rules and discuss how the problem was solved. Our guideline when we devised the transformations was that each transition in the proof tree should correspond to a step in a rewriting sequence and that the rewriting sequences should mimic the left-to-right construction of proof trees.

5.1.1. Original Rules

The 2BIG rules below define the evaluation of sum expressions. The state S is not used in the example, but if we allow variables in sum expressions, then we could add a rule like $\frac{}{\mathbf{var}(X) \triangleright S \to lookup(X,S)}$ for variables whose values are stored somewhere in the state.

$$\frac{}{\mathbf{num}(N) \triangleright S \to N} \qquad \frac{E_1 \triangleright S \to V_1 \quad E_2 \triangleright S \to V_2}{\mathbf{add}(E_1, E_2) \triangleright S \to plus(V_1, V_2)}$$

A proof tree for **add(num(1),add(num(2),num(3)))** is shown below

$$\dfrac{\dfrac{}{\mathbf{num}(\mathbf{1}) \triangleright nil \to 1} \quad \dfrac{\dfrac{}{\mathbf{num}(\mathbf{2}) \triangleright nil \to 2} \quad \dfrac{}{\mathbf{num}(\mathbf{3}) \triangleright nil \to 3}}{\mathbf{add}(\mathbf{num}(\mathbf{2}), \mathbf{num}(\mathbf{3})) \triangleright nil \to 5\boxed{1}}}{\mathbf{add}(\mathbf{num}(\mathbf{1}), \mathbf{add}(\mathbf{num}(\mathbf{2}), \mathbf{num}(\mathbf{3}))) \triangleright nil \to 6\boxed{2}}$$

The results of evaluating functions in the above proof tree are marked with boxed numbers, in particular $\boxed{1}$ is the result of $plus(2,3)$ and $\boxed{2}$ the result of $plus(1,5)$. Consider the process of building this proof tree from left to right. We make two observations:

Observation 5.1.

(i) After building the trees $\frac{}{\mathbf{num}(\mathbf{2}) \triangleright nil \to 2}$ and $\frac{}{\mathbf{num}(\mathbf{3}) \triangleright nil \to 3}$ we can evaluate $plus(2,3)$. Here 2 is taken from the first proof tree and 3 from the second.

(ii) Furthermore *nil* is not contained in the right hand side of the conclusion in the proof tree of the first precondition, but is used again in the left hand side of the conclusion in the proof tree of the second precondition.

In contrast in a sequence of term rewriting steps $t_1 \overset{1}{\Rightarrow} \ldots \overset{1}{\Rightarrow} t_n$ all values needed for a rewrite step $t_i \overset{1}{\Rightarrow} t_{i+1}$

must be present in t_i. Since each transition in the proof tree should correspond to a step in a rewriting sequence using the generated term rewriting rules, we have to modify the transitions without changing the semantics. The following transformation achieves that all values are passed from transition to transition until they are used for the last time.

5.1.2. Allocation of Temporary Variables

Now consider the following rules which result from a transformation that we call 'allocation of temporary variables'. It adds a new component D to the state and puts temporary variables there.

$$\frac{}{\mathbf{num}(N) \triangleright [D, S] \to [D, N]} \qquad \frac{E_1 \triangleright [[[S]|D], S] \to [[[S]|D], V_1] \quad E_2 \triangleright [[[V_1]|D], S] \to [[[V_1]|D], V_2]}{\mathbf{add}(E_1, E_2) \triangleright [D, S] \to [D, plus(V_1, V_2)]}$$

A proof tree for **add(num(1),add(num(2),num(3)))** is shown below.

$$\dfrac{\dfrac{}{\mathbf{num(1)} \triangleright [[[nil]], nil] \to [[[nil]], 1]} \quad \dfrac{\dfrac{}{\mathbf{num(2)} \triangledown [[[nil], [1]], nil] \downarrow [[[nil], [1]], 2]\,\boxed{1}} \quad \dfrac{}{\mathbf{num(3)} \triangledown [[[2], [1]], nil]\,\boxed{2} \downarrow [[[2], [1]], 3]\,\boxed{3}}}{\mathbf{add(num(2),num(3))} \triangleright [[[1]], nil] \to [[[1]], 5]\,\boxed{4}}}{\mathbf{add(num(1), add(num(2), num(3)))} \triangleright [[], nil] \to [[], 6]}$$

When we build the new proof tree from left to right we first build $\overline{\mathbf{num(2)} \triangleright [[[nil], [1]], nil] \to [[[nil], [1]], 2]}$ and then $\overline{\mathbf{num(3)} \triangleright [[[2], [1]], nil] \to [[[2], [1]], 3]}$. It turns out that the two points we made in Observation 5.1 are not true for this tree:

Observation 5.2.

(i) revisited: The values 2 and 3 are both contained on the right hand side of the conclusion in the second proof tree.

(ii) revisited: *nil* occurs on the right hand side of the conclusion in the first proof tree.

In the above proof tree we marked certain states with boxed numbers. For these states we make the following observations:

(iii) The state $\boxed{1}$ is different from the state $\boxed{2}$.

(iv) The state $\boxed{3}$ differs from the state $\boxed{4}$.

Now look again at a sequence of term rewriting steps $t_1 \overset{1}{\Rightarrow} \ldots \overset{1}{\Rightarrow} t_n$. Here the term t_{i+1} which 'results' from the rewriting step $t_i \overset{1}{\Rightarrow} t_{i+1}$ is the 'input' to the next rewriting step $t_{i+1} \overset{1}{\Rightarrow} t_{i+2}$.

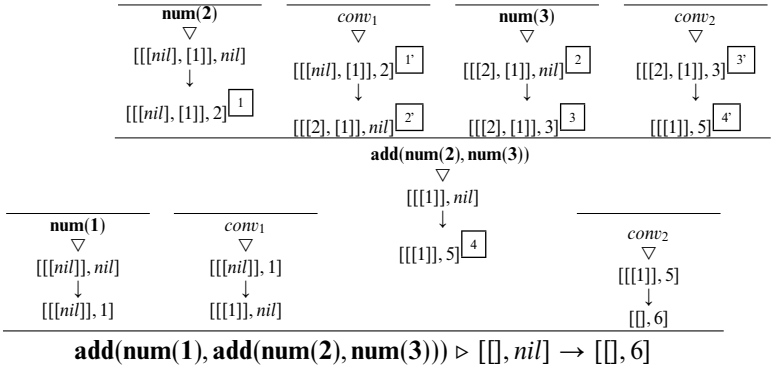
5.1.3. Sequentialisation of Rules

Next we add transitions to the rules which convert the resulting state of a transitition into the form needed for its following transition. To this end we introduce two new instructions $conv_1$ and $conv_2$.

$$\frac{\begin{array}{ll} E_1 \triangleright [[[S]|D], S] \to [[[S]|D], V_1] & conv_1 \triangleright [[[S]|D], V_1] \to [[[V_1]|D], S] \\ E_2 \triangleright [[[V_1]|D], S] \to [[[V_1]|D], V_2] & conv_2 \triangleright [[[V_1]|D], V_2] \to [D, plus(V_1, V_2)] \end{array}}{\mathbf{add}(E_1, E_2) \triangleright [D, S] \to [D, plus(V_1, V_2)]}$$

$$\overline{\mathbf{num}(N) \triangleright [D, S] \to [D, N]} \quad \overline{conv_1 \triangleright [[[S]|D], V] \to [[[V]|D], S]} \quad \overline{conv_2 \triangleright [[[V_1]|D], V_2] \to [D, plus(V_1, V_2)]}$$

Using the sequentialised rules we obtain the proof tree:



Reconsidering Observation 5.2 for our previous proof tree we can make the following observation.

Observation 5.3.

(iii) revisited: The state [1] is equal to [1'] which is converted to [2'] and [2'] is equal to [2].

(iv) revisited: The state [3] is equal to [3'] which is converted to [4'] and [4'] is equal to [4].

From these rules we can now generate a term rewriting system.

5.1.4. Term Rewriting Systems

Before we describe the next transformation which generates term rewriting rules, we provide here the relevant definitions; they are based on those given in [Han94, Jou95].

Definition 5.1 (Term Rewriting System). Let Σ be a **signature**, i.e., a finite set of constants, and X be a set of variables. $T_\Sigma(X)$ is the set of all first-order terms with variables in X. A **term rewriting system** is a pair (Σ, R), where R is a set of rules $l_i \Rightarrow r_i$ with $l_i, r_i \in T_\Sigma(X)$ and $\mathcal{V}(r_i) \subseteq \mathcal{V}(l_i)$. Furthermore a rule $l_i \Rightarrow r_i$ is **linear** (left-linear) if no variable occurs twice in l_i.

For example the term rewriting system (Σ_+, R_+) where $\Sigma_+ = \{null, s, +\}$
and $R_+ = \left\{ \begin{array}{l} null + N \Rightarrow N, \\ s(N) + M \Rightarrow N + s(M) \end{array} \right\}$ defines addition on a unary encoding of positive natural numbers.

Let s be a term, $(l \Rightarrow r) \in R$ and θ be a substitution such that $\theta(l) = s$, then $s \overset{1}{\Rightarrow}_R \theta(r)$ is a one-step reduction. If there exists a sub-term s in t such that $s \overset{1}{\Rightarrow}_R s'$ and replacing one occurrence of s by s' in t yields u, then $t \Rightarrow_R u$ is a single rewrite step. We write $\overset{*}{\Rightarrow}_R$ for the reflexive, transitive closure of $\Rightarrow_R$.

Returning to the above example, addition of the two numbers 2 and 1 using R_+ leads to the following rewriting sequence: $s(s(null))+s(null) \overset{1}{\Rightarrow}_{R_+} s(null)+s(s(null)) \overset{1}{\Rightarrow}_{R_+} null+s(s(s(null))) \overset{1}{\Rightarrow}_{R_+} s(s(s(null)))$ and thus we have $s(s(null))+s(null) \overset{*}{\Rightarrow}_{R_+} s(s(s(null)))$.

5.1.5. Properties of Term Rewriting Systems

Definition 5.2 (Normal form, termination, confluence, orthogonality). A term t is in R-**normal form**, iff there exists no term u such that $t \Rightarrow_R u$. If $s \overset{*}{\Rightarrow}_R u$ and u is in normal form, then we write $s \overset{\dagger}{\Rightarrow}_R u$. R is **terminating** iff there is no infinite sequence $t_1 \Rightarrow_R t_2 \Rightarrow_R \ldots$. R is **confluent** iff ($t \overset{*}{\Rightarrow}_R t_1$ and $t \overset{*}{\Rightarrow}_R t_2$) implies that a term t' exists such that $t_1 \overset{*}{\Rightarrow}_R t'$ and $t_2 \overset{*}{\Rightarrow}_R t'$. R is **overlapping** if a left hand side unifies with a renamed non-variable subterm of some other left hand side or with a renamed proper subterm of itself. R is **orthogonal** if it is both linear and non-overlapping.

Theorem 5.4 (proof in [Hue80]). Every orthogonal system is confluent.

In practice, functions (e.g., $+$, $-$, ...) are often used in term rewriting rules without specifying rewrite rules for these functions. Alternatively, we will use in the proofs the following extension of term rewriting by

redefining one-step reductions: Let s be a term, $(l \Rightarrow r) \in R$ and θ be a substitution such that $\theta(l) = s$, then $s \overset{1}{\Rightarrow}_R \eta_\sigma(\theta(r))$ is a one-step reduction. The evaluation function η_σ was defined in Section 2.3.2 (Definition 2.7).

5.1.6. Conversion into Term Rewriting System

In the generated set I of term rewriting rules, terms are of the form $\langle c, s\rangle$ where c corresponds to the list of goals which have to be proved and s is the state which the first goal in the list has to be proved in. We call these goals instructions and the list of goals, which have to be proved, the program.

As an example look at the 2BIG rule for $\mathbf{add}(E_1, E_2)$ after sequentialisation in Section 5.1.3. To prove a transition for this expression in state $[D, S]$, we have to prove transitions for $E_1, conv_1, E_2$ and $conv_2$. Moreover the transition for E_1 has to be proved in the state $[[[S]|D], S]$. Putting this together we get the term rewriting rule $\langle \mathbf{add}(E_1, E_2); C, [D, S]\rangle \Rightarrow_I \langle E_1; conv_1; E_2; conv_2; C, [[[S]|D], S]\rangle$. Applying this conversion to all rules we get:

$$
\begin{array}{lll}
\langle \mathbf{num}(N); C, [D, S]\rangle & \Rightarrow_I & \langle C, [D, N]\rangle \\
\langle \mathbf{add}(E_1, E_2); C, [D, S]\rangle & \Rightarrow_I & \langle E_1; conv_1; E_2; conv_2; C, [[[S]|D], S]\rangle \\
\langle conv_1; C, [[[S]|D], V]\rangle & \Rightarrow_I & \langle C, [[[V]|D], S]\rangle \\
\langle conv_2; C, [[[V_1]|D], V_2]\rangle & \Rightarrow_I & \langle C, [D, plus(V_1, V_2)]\rangle
\end{array}
$$

We can use these term rewriting rules to compute the value of our example expression. We use **nop** for the empty instruction sequence here. Alternatively one could regard it as the special instruction which does nothing.

$$
\begin{array}{lll}
 & \langle \mathbf{add}(\mathbf{num}(\mathbf{1}), \mathbf{add}(\mathbf{num}(\mathbf{2}), \mathbf{num}(\mathbf{3}))); \mathbf{nop}, & [[], nil]\rangle \\
\Rightarrow & \langle \mathbf{num}(\mathbf{1}); conv_1; \mathbf{add}(\mathbf{num}(\mathbf{2}), \mathbf{num}(\mathbf{3})); conv_2; \mathbf{nop}, & [[[nil]], nil]\rangle \\
\Rightarrow & \langle conv_1; \mathbf{add}(\mathbf{num}(\mathbf{2}), \mathbf{num}(\mathbf{3})); conv_2; \mathbf{nop}, & [[[nil]], 1]\rangle \\
\Rightarrow & \langle \mathbf{add}(\mathbf{num}(\mathbf{2}), \mathbf{num}(\mathbf{3})); conv_2; \mathbf{nop}, & [[[1]], nil]\rangle \\
\Rightarrow & \langle \mathbf{num}(\mathbf{2}); conv_1; \mathbf{num}(\mathbf{3}); conv_2; conv_2; \mathbf{nop}, & [[[nil], [1]], nil]\rangle \\
\Rightarrow & \langle conv_1; \mathbf{num}(\mathbf{3}); conv_2; conv_2; \mathbf{nop}, & [[[nil], [1]], 2]\rangle \\
\Rightarrow & \langle \mathbf{num}(\mathbf{3}); conv_2; conv_2; \mathbf{nop}, & [[[2], [1]], nil]\rangle \\
\Rightarrow & \langle conv_2; conv_2; \mathbf{nop}, & [[[2], [1]], 3]\rangle \\
\Rightarrow & \langle conv_2; \mathbf{nop}, & [[[1]], 5]\rangle \\
\Rightarrow & \langle \mathbf{nop}, & [[], 6]\rangle
\end{array}
$$

5.1.7. Generated Definition of a Compiler and Abstract Machine

Consider the term rewriting rule for **add**. When we apply this rule both the program and the state are modified. But the modification of the program does not depend on a variable in the state. As a consequence the modifications can be done independently and thus at different times. The modification of the program can be done at compile-time and the modification of the state is deferred until run-time. A new instruction $\overline{add}$ is introduced which does the deferred modification. Those instructions which do not change the program other than removing its first instruction become instructions of the abstract machine. The same holds for instructions for which the modification of the program depends on the state. To indicate that an instruction is an abstract machine instruction we put a line over its constructor. Thus we get the following set C of rules defining a compiler:

$$
\mathbf{num}(N) \Rightarrow_C \overline{num}(N) \qquad \mathbf{add}(E_1, E_2) \Rightarrow_C \overline{add}; E_1; \overline{conv}_1; E_2; \overline{conv}_2
$$

and the following set X of rules defining an abstract machine:

$$
\begin{array}{ll}
\langle \overline{num}(N); C, [D, S]\rangle \Rightarrow_X \langle C, [D, N]\rangle & \langle \overline{conv}_1; C, [[[S]|D], V]\rangle \Rightarrow_X \langle C, [[[V]|D], S]\rangle \\
\langle \overline{add}; C, [D, S]\rangle \Rightarrow_X \langle C, [[[S]|D], S]\rangle & \langle \overline{conv}_2; C, [[[V_1]|D], V_2]\rangle \Rightarrow_X \langle C, [D, plus(V_1, V_2)]\rangle
\end{array}
$$

Compilation of **add(num(1),add(num(2),num(3)))** yields

$$
\overline{add}; \overline{num}(1); \overline{conv}_1; \overline{add}; \overline{num}(2); \overline{conv}_1; \overline{num}(3); \overline{conv}_2; \overline{conv}_2
$$

Now we can use these abstract machine rules to compute the value of the compiled expression.

$$
\begin{array}{ll}
 & \langle \overline{add}; \overline{num}(1); \overline{conv}_1; \overline{add}; \overline{num}(2); \overline{conv}_1; \overline{num}(3); \overline{conv}_2; \overline{conv}_2; nop, & [[], nil]\rangle \\
\Rightarrow & \langle \overline{num}(1); \overline{conv}_1; \overline{add}; \overline{num}(2); \overline{conv}_1; \overline{num}(3); \overline{conv}_2; \overline{conv}_2; nop, & [[[nil]], nil]\rangle \\
\Rightarrow & \langle \overline{conv}_1; \overline{add}; \overline{num}(2); \overline{conv}_1; \overline{num}(3); \overline{conv}_2; \overline{conv}_2; nop, & [[[nil]], 1]\rangle \\
\Rightarrow & \langle \overline{add}; \overline{num}(2); \overline{conv}_1; \overline{num}(3); \overline{conv}_2; \overline{conv}_2; nop, & [[[1]], nil]\rangle \\
\Rightarrow & \langle \overline{num}(2); \overline{conv}_1; \overline{num}(3); \overline{conv}_2; \overline{conv}_2; nop, & [[[nil], [1]], nil]\rangle \\
\Rightarrow & \langle \overline{conv}_1; \overline{num}(3); \overline{conv}_2; \overline{conv}_2; nop, & [[[nil], [1]], 2]\rangle \\
\Rightarrow & \langle \overline{num}(3); \overline{conv}_2; \overline{conv}_2; nop, & [[[2], [1]], nil]\rangle \\
\Rightarrow & \langle \overline{conv}_2; \overline{conv}_2; nop, & [[[2], [1]], 3]\rangle \\
\Rightarrow & \langle \overline{conv}_2; nop, & [[[1]], 5]\rangle \\
\Rightarrow & \langle nop, & [[], 6]\rangle
\end{array}
$$

Note that this rewriting sequence has the same length as the rewriting sequence we obtain using the original term rewriting rules. Furthermore, the state in the i-th term in one sequence is equal to the state in the i-th term in the other. The major difference to the original sequence is that in this abstract machine at each rewrite step an instruction is removed from the program. When it comes to implementing the abstract machine, this allows us to store the program in an array and use a program counter which points to the instruction that has to be executed. After execution of the instruction, the program counter is increased by one.

5.2. Generating Compilers and Abstract Machines from 2BIG Specifications

An overview of the system is given in Fig. 2. Since the system transforms specifications by successively applying transformations, we will present the transformations in the order of their application. Actually the transformations have been devised in reverse order. Starting from the pass separation transformation, we tried to remove restrictions on the input specifications by transforming a more general class of specifications into the class of input specifications. This process finally leads to determinate 2BIG specifications. Note that after each transformation we have an executable specification again.

We will use the following excerpts of a 2BIG semantics for an imperative language as a running example:

$$\frac{B \triangleright S \to true \quad C; \textbf{while } B \textbf{ do } C \textbf{ od} \triangleright S \to S'}{\textbf{while } B \textbf{ do } C \textbf{ od} \triangleright S \to S'} \qquad \frac{B \triangleright S \to false}{\textbf{while } B \textbf{ do } C \textbf{ od} \triangleright S \to S}$$

$$\frac{B \triangleright S \to V \quad is\_bool(V)}{\textbf{bool}(B) \triangleright S \to V} \qquad \frac{B \triangleright S \to V \quad not(is\_bool(V))}{\textbf{bool}(B) \triangleright S \to \textbf{type\_error}}$$

5.2.1. Source Variables

Compile-time objects are those which can be constructed or evaluated at compile-time using the information in the program only without having to refer to information in the state or input to the program.

Consider the first rule of the example 2BIG specification of **while**. Here B and C are bound to subprograms known at compile-time, whereas S and S' are run-time data. Because B and C are known at compile-time, the arguments to the sequencing instruction ; (written as an infix operator) of the precondition C; **while** B **do** C **od** $\triangleright$ $S \to S'$, are also known at compile-time. In the proof tree for **while i** $>$ **1 do i** $:=$ **i** $-$ **1 od** $\triangleright$ [**i** $\mapsto$ 2] $\to$ [**i** $\mapsto$ 1] at the end of Section 4, there is no term on the left of $\triangleright$, which was not present in the original program **while i** $>$ **1 do i** $:=$ **i** $-$ **1 od**.

Because of the important role they play in our transformations, we define two kinds of variables based on their first defining occurrence in a rule:

Definition 5.3 (Source Variable, Input State Variable). Let $\frac{s_1 \quad \cdots \quad s_n}{c \triangleright e \to e'}$ be a 2BIG rule, then we call a variable $x \in \mathscr{V}(c)$ a **source variable** and a variable $y \in \mathscr{V}(e)$ an **input state variable**.

Note that because we require 2BIG rules to be linear, a variable can not be both a source and an input state variable.

In general, if all rules have the property that arguments to instructions in the preconditions contain only

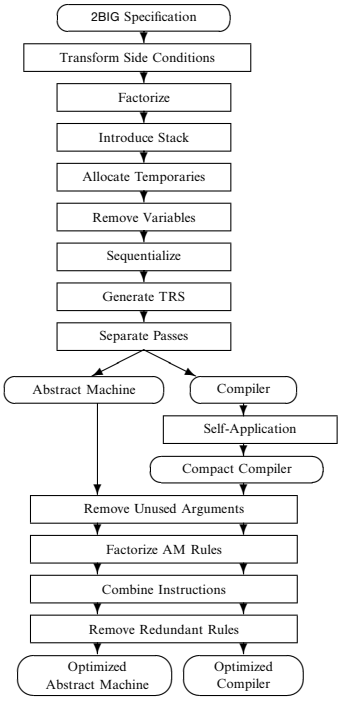


Fig. 2. Overview of the transformations.

source variables then we can only build proof trees for $c \triangleright e \rightarrow e'$ where all variables on the left of $\triangleright$ get bound to terms occurring in c.

But there are two problems with this restriction. First, even if all rules have the required form, we do not know how often the rules are applied, i.e., we know at compile-time what terms occur on the left of $\triangleright$, but we do not know the size of the proof tree. Second, in many specifications of programming languages, the definitions of procedures or functions are contained in the environment, which is part of the state and applying the function or executing the procedure is expressed by moving the definition from the input state, i.e., from the right of $\triangleright$, to the instruction, i.e., to the left of $\triangleright$, of a precondition.

For example, an expression stored on top of a stack can be reduced by executing it as in

$$\frac{E \triangleright S \rightarrow S'}{apply \triangleright [E|S] \rightarrow S'}$$

One of the main rationales in our transformations is to split run-time and compile-time data. Compile-time data have to be passed as arguments of instructions. Run-time data become components of the state. At the end, we want to compile a source language program, which is an instruction without variables, into a sequence of abstract machine instructions. In this machine program, no run-time data must occur. Compiler rules cannot access any run-time data, thus all arguments to abstract machine instructions used in compiler rules must be constant terms or terms which occurred in the source program. Thus every time one of our transformations introduces a new instruction, it makes sure that the arguments to that instruction are source variables.

5.2.2. Transformation of Side Conditions

As we mentioned before, we regard side conditions as syntactic sugar for special kinds of transitions. In semantic specifications side conditions are used to express tests on variables in a rule. More precisely, with every predicate p we associate an n-ary relation Π on terms. There exists a proof for a side condition $p(a_1, \ldots, a_n)$, if $(a_1, \ldots, a_n) \in \Pi$.

The characteristic function χ_p is defined as $\chi_p(x) = \begin{cases} true & if\ x \in \Pi \\ false & if\ x \notin \Pi \end{cases}$ Using characteristic functions, side conditions can be converted into transitions. Let $\frac{s_1 \ \ldots \ s_j}{c \rhd e \to e'}$ be a rule. If s_i is a side condition of the form $p(t_1, \ldots, t_n)$, then it is converted into a transition $a(x_1, \ldots, x_k) \rhd [y_1, \ldots, y_m] \to true$.

A side condition of the form *not* $p(t_1, \ldots, t_n)$ is converted into a transition $a(x_1, \ldots, x_k) \rhd [y_1, \ldots, y_m] \to false$, where a is a new instruction symbol, for which we generate a new rule $\frac{}{a(x_1,\ldots,x_k) \rhd [y_1,\ldots,y_m] \to \chi_p(t_1,\ldots,t_n)}$ and χ_p is the characteristic function of the predicate p, $\{x_1, \ldots, x_k\} = \mathcal{V}(c) \cap \mathcal{V}(t_1, \ldots, t_n)$ and $\{y_1, \ldots, y_m\} = \mathcal{V}(t_1, \ldots, t_n) - \mathcal{V}(c)$. Dividing the variables occurring in the side condition this way guarantees that only source variables $x_1, \ldots, x_k$ are arguments of the new instruction. The remaining variables $y_1, \ldots, y_m$ are passed in the state. To reduce the number of generated instructions, we identify two generated instructions if their defining rules are equal modulo the instruction symbols.

Applying the above transformation to the rules for **bool**(B) we get

$$\frac{B \rhd S \to V \quad test \rhd [V] \to true}{\mathbf{bool}(B) \rhd S \to V} \qquad \frac{B \rhd S \to V \quad test \rhd [V] \to false}{\mathbf{bool}(B) \rhd S \to \mathbf{type\_error}} \qquad \frac{}{test \rhd [V] \to is\_bool(V)}$$

5.2.3. *Factorisation*

The analogy between natural semantics and grammars has already been noted by G. Kahn [Kah87]. Grammar rules define legal parse trees and as a consequence legal sentences. In analogy, inference rules define legal proof trees and as a consequence derivable facts. Some grammars can be converted by an algorithm called left-factoring [AhU72, ASU86, LeP81] into a form that has the property that a top-down parser can decide among rules for the same nonterminal by looking at the next input symbol. This class of grammars is usually called LL(1). Let $A \to \alpha\beta_1 | \ldots | \alpha\beta_n | \gamma_1 | \ldots | \gamma_m$ be the grammar rules for the nonterminal A. Grammar rules for the same nonterminal which have a common prefix α are replaced by a single rule $A \to \alpha A'$ and a new nonterminal A' is introduced which produces the suffixes $\beta_1, \ldots, \beta_n$. Thus we get the new rules $A \to \alpha A' | \gamma_1 | \ldots | \gamma_m$ and $A' \to \beta_1 | \ldots | \beta_n$. This transformation is repeated as long as there is a common prefix in the rules for a nonterminal.

The transformation presented in this section has much in common with the above algorithm, but note that there is a crucial difference between grammars and inference rules. When applying a grammar rule, occurrences of the same nonterminal in a rule can produce different sentences, whereas when we apply an inference rule, occurrences of the same variable have to be substituted by the same term.

Factorisation of inference rules converts sets of determinate inductive rules into deterministic rules. Fabio daSilva [daS90] developed a factorisation transformation that has been proved correct for the case of sets with two conflicting rules. We extend his transformation to sets of more than two conflicting rules. Basically the transformation generates for a set of n conflicting rules $n + 1$ new rules. First a rule is generated which has as its preconditions the common initial preconditions of the conflicting rules and a transition with a new instruction. To define the new instruction, the transformation generates for each of the n conflicting rules a rule which has the rest of the preconditions of the conflicting rule as its preconditions. We repeat factorisation until there are no more conflicting rules. By $=_\alpha$ we will denote equality of terms and formulae modulo renaming of variables.

Definition 5.4 (Conflicting Rules). Two rules are **conflicting**, if they have the same left hand sides in their conclusions.

Let $\mathscr{C}$ be the largest set of conflicting rules with respect to the same left hand side (*When left-factoring grammars this set is analogous to the set of grammar rules for the same nonterminal*):

$$\frac{c_{11} \rhd e_{11} \to e'_{11} \quad \ldots \quad c_{1m_1} \rhd e_{1m_1} \to e'_{1m_1}}{c_1 \rhd e_1 \to e'_1} \quad \cdots \quad \frac{c_{n1} \rhd e_{n1} \to e'_{n1} \quad \ldots \quad c_{nm_n} \rhd e_{nm_n} \to e'_{nm_n}}{c_n \rhd e_n \to e'_n}$$

where $c_1, e_1 =_\alpha \ldots =_\alpha c_n, e_n$.

Let θ be a renaming of variables and j be the largest integer, such that for all $p, q \in \{1, \ldots, n\}$: $(c_{pj}, e_{pj})\theta = (c_{qj}, e_{qj})\theta$ and

$$\forall k < j : (c_{pk}, e_{pk}, e'_{pk})\theta = (c_{qk}, e_{qk}, e'_{qk})\theta \tag{1}$$

It should be kept in mind that by the renaming θ the terms are α-equal. This is a stronger property than unifiability. Let us call the ordered set of the latter transitions the **common initial segment** *seg*. More precisely,

the common initial segment seg_i in the i-th rule is the ordered set of the first $j-1$ preconditions of that rule. We arbitrarily choose $seg = seg_1$. (*When left-factoring grammars the common segment is analogous to the common prefix*).

The common term of two terms is the most general term modulo renaming of variables which unifies with both terms. We define the **common term** $e_1 \odot_\tau e_2$ of two terms e_1 and e_2 with respect to a variable renaming τ as:

$$e_1 \odot_\tau e_2 = \begin{cases} e_1\tau & \text{if } e_1\tau = e_2\tau \text{ are the same variable name} \\ f(d_1,\ldots,d_n) & \text{if } e_1 = f(a_1,\ldots,a_n),\ e_2 = f(b_1,\ldots,b_n) \\ & \text{and } d_i = a_i \odot_\tau b_i \\ c(d_1,\ldots,d_n) & \text{if } e_1 = c(a_1,\ldots,a_n),\ e_2 = c(b_1,\ldots,b_n) \\ & \text{and } d_i = a_i \odot_\tau b_i \\ x & \text{otherwise} \end{cases}$$

where x is a new variable name.

Note, that $\odot$ is associative and commutative. Thus there is also a unique common term for more than two terms. Now let $e^\bullet$ be the common term of $e'_{1j},\ldots,e'_{nj}$ with respect to θ, i.e., $e^\bullet = e'_{1j} \odot_\theta \ldots \odot_\theta e'_{nj}$; furthermore let ι be a new instruction symbol and

$$\begin{array}{l} \text{Variables used in the remaining preconditions:} \\ \quad \mathscr{R}_1 = \bigcup_{k=1}^{n} \mathscr{V}(c_{k(j+1)},\ldots,c_{km_k},e_{k(j+1)},\ldots,e_{km_k},e'_k)\theta \\ \text{Variables defined in } seg\text{, the conclusion or in } c_{kj}, e_{kj}\text{:} \\ \quad \mathscr{R}_2 = \mathscr{V}(seg,c_1,e_1)\theta \cup \bigcup_{k=1}^{n} \mathscr{V}(c_{kj},e_{kj})\theta \\ \text{Variables used in the remaining preconditions, defined in } seg\text{, etc.} \\ \text{and not passed in the common term } e^\bullet\text{:} \\ \quad \mathscr{R}_3 = (\mathscr{R}_1 \cap \mathscr{R}_2) - \mathscr{V}(e^\bullet)\theta \\ \text{Those variables in } \mathscr{R}_3 \text{ which are not source variables:} \\ \quad \mathscr{R} = \mathscr{R}_3 - \mathscr{V}(\theta(c_1)) \\ \text{Source variables are passed as arguments:} \\ \quad \kappa = \iota(x_1,\ldots,x_m) \quad \text{where } x_i \in \mathscr{R}_3 \cap \mathscr{V}(\theta(c_1)) \end{array} \tag{2}$$

Above we computed those variables which have to be passed to the rules of the newly created instruction. Some of them are passed in κ, some in R and some in $e^\bullet$. Let e' be a new variable name, then $\mathscr{C}$ is replaced by:

$$\frac{seg \quad c_{1j} \rhd e_{1j} \to e^\bullet \quad \kappa \rhd [\mathscr{R}, e^\bullet] \to e'}{c_1 \rhd e_1 \to e'}\theta$$

$$\frac{c_{1(j+1)} \rhd e_{1(j+1)} \to e'_{1(j+1)} \quad \ldots \quad c_{1m_1} \rhd e_{1m_1} \to e'_{1m_1}}{\kappa \rhd [\mathscr{R}, e'_{1j}] \to e'_1}\theta$$

$$\vdots$$

$$\frac{c_{n(j+1)} \rhd e_{n(j+1)} \to e'_{n(j+1)} \quad \ldots \quad c_{nm_n} \rhd e_{nm_n} \to e'_{nm_n}}{\kappa \rhd [\mathscr{R}, e'_{nj}] \to e'_n}\theta$$

When left-factoring grammars a new nonterminal is introduced, which produces the different suffixes of the grammar rules. Here we have to introduce a new instruction and via the arguments to that instruction and the state we have to ensure that it allows to prove exactly those remaining non-α-equal preconditions of the original rules.

For the *while*-loop rules, we get that $seg = \emptyset$, $e^\bullet = true \odot false = Y$ where Y is a new variable, $\mathscr{R}_1 = \{C,B,S,S'\}$, $\mathscr{R}_3 = \mathscr{R}_2 = \{C,B,S\}$, $\mathscr{R} = \{S\}$, $\kappa = factor(C,B)$ and thus factorisation of the semantics rules yields:

$$\frac{B \rhd S \to Y \quad factor_1(C,B) \rhd [[S],Y] \to S'}{\textbf{while } B \textbf{ do } C \textbf{ od} \rhd S \to S'} \qquad \frac{}{test \rhd [V] \to is\_bool(V)}$$

$$\frac{C; \textbf{while } B \textbf{ do } C \textbf{ od} \rhd S \to S'}{factor_1(C,B) \rhd [[S],true] \to S'} \qquad \frac{}{factor_1(C,B) \rhd [[S],false] \to S}$$

$$\frac{B \rhd S \to V_1 \quad test \rhd [V_1] \to R \quad factor_2 \rhd [[V_1],R] \to V_2}{\textbf{bool}(B) \rhd S \to V_2}$$

$$\frac{}{factor_2 \rhd [[V],false] \to \textbf{type\_error}} \qquad \frac{}{factor_2 \rhd [[V],true] \to V}$$

5.2.4. Stack Introduction

In the next step, the states in the rules are extended by a stack. This stack will be used later to store temporary variables.[4] In our examples we use the variable D for the stack or dump, because S is already used for the store. The transformation converts rules of the form $\frac{c_1 \triangleright e_1 \to e'_1 \quad \ldots \quad c_n \triangleright e_n \to e'_n}{c \triangleright e \to e'}$ into $\frac{c_1 \triangleright [s,e_1] \to [s,e'_1] \quad \ldots \quad c_n \triangleright [s,e_n] \to [s,e'_n]}{c \triangleright [s,e] \to [s,e']}$ where s is a new variable name.

Adding a stack D to the first rule of the *while*-loop yields:

$$\frac{B \triangleright [D,S] \to [D,Y] \qquad factor_1(C,B) \triangleright [D,[[S],Y]] \to [D,S']}{\textbf{while } B \textbf{ do } C \textbf{ od} \triangleright [D,S] \to [D,S']}$$

5.2.5. Allocation of Temporary Variables

A variable is called temporary in a rule, if there is an intermediate state where the variable does not occur and it is not a source variable. Our goal is that source variables are passed from transition to transition in instructions whereas temporary variables are passed in the state. Furthermore we do not allocate anonymous variables, i.e., variables, which only occur once in a rule. The rules are transformed, such that temporary variables are passed in the state from the precondition of their first occurrence to the precondition of their last occurrence.

In the following definition we use the straightforward extension of $\mathcal{V}(\,)$ to sets of terms.

Definition 5.5 (Temporary Variable). A variable x is **temporary** in a rule $(\{c_1 \triangleright e_1 \to e'_1, \ldots, c_n \triangleright e_n \to e'_n\}, c \triangleright e \to e')$, if $x \notin \mathcal{V}(c)$ and there is an i such that

1. $x \in \mathcal{V}(c_i, e_i)$, and $x \in \mathcal{V}(e, c_1, e_1, e'_1, \ldots, c_{i-1}, e_{i-1})$, but $x \notin \mathcal{V}(e'_{i-1})$
2. or $x \in \mathcal{V}(e'_i)$ and $x \in \mathcal{V}(e, c_1, e_1, e'_1, \ldots, c_{i-1}, e_{i-1}, e'_{i-1})$, but $x \notin \mathcal{V}(c_i, e_i)$
3. or $x \in \mathcal{V}(e')$ and $x \in \mathcal{V}(e, c_1, e_1, e'_1, \ldots, c_n, e_n)$, but $x \notin \mathcal{V}(e'_n)$

Definition 5.6 (Allocated Rules). A set ϕ of 2BIG rules is **allocated**, iff $r \in \phi \Rightarrow$ there is no temporary variable in r.

Definition 5.7 (Computation of Temporary Variables).
Let $\frac{c_1 \triangleright e_1 \to e'_1 \quad \ldots \quad c_n \triangleright e_n \to e'_n}{c \triangleright e \to e'}$ be a 2BIG rule, then we define the sets $\mathcal{M}_j$ $(1 \leqslant j \leqslant n)$ of variables temporary in the precondition with index j:

$$\begin{array}{ll} \text{for } 1 \leqslant i < n: & \\ \mathcal{M}_i & = \{x | x \notin \mathcal{V}(c), x \notin \mathcal{V}(e'_i), x \in \mathcal{V}(c_{i-1}, e_{i-1}), x \in \mathcal{V}(e, c_1, e_1, e'_1, \ldots, c_i, e_i)\} \\ & \cup \{x | x \notin \mathcal{V}(c), x \notin \mathcal{V}(c_i, e_i), x \in \mathcal{V}(e'_i), x \in \mathcal{V}(e, c_1, e_1, e'_1, \ldots, c_{i-1}, e_{i-1}, e'_{i-1})\} \\ \mathcal{M}_n & = \{x | x \notin \mathcal{V}(c), x \notin \mathcal{V}(e'_n), x \in \mathcal{V}(e'), x \in \mathcal{V}(e, c_1, e_1, e'_1, \ldots, c_n, e_n)\} \end{array}$$

Theorem 5.5. A variable x is **temporary** in a rule if there is an i such that $x \in \mathcal{M}_i$.

Now consider the rules resulting from stack introduction:

$$\frac{c_1 \triangleright [s_1, e_1] \to [s'_1, e'_1] \quad \ldots \quad c_n \triangleright [s_n, e_n] \to [s'_n, e'_n]}{c \triangleright [s, e] \to [s', e']}$$

We convert the preconditions in the rule as follows: Let $c_k \triangleright [s_k, e_k] \to [s'_k, e'_k]$ be the k-th precondition in the rule. Then it is converted into $c_k \triangleright [[\overline{\mathcal{M}_k} | s_k], e_k] \to [[\overline{\mathcal{M}_k} | s'_k], e'_k]$ where $\overline{\mathcal{M}_k}$ is a list of all variables in $\mathcal{M}_k$. If $\mathcal{M}_k$ is empty, then we do not change the precondition. Note that allocating temporary variables before factorisation would destroy common initial segments. Consider the two 2BIG rules for **bool**(B) after transformation of side conditions. The left hand sides of the second precondition of both rules are equal and would be part of the common segment. In the first rule V is temporary; in the second rule there is no temporary variable. Thus we would get:

[4] A possible optimisation not discussed here stores the results of function calls on the stack.

$$\frac{B \triangleright [D,S] \to [D,V] \quad test \triangleright [[[V]|D],[V]] \to [[[V]|D], true]}{\mathbf{bool}(B) \triangleright [D,S] \to [D,V]}$$

$$\frac{B \triangleright [D,S] \to [D,V] \quad test \triangleright [D,[V]] \to [D, false]}{\mathbf{bool}(B) \triangleright [D,S] \to [D, \mathbf{type\_error}]}$$

Allocating the variable V in the first rule but not in the second changes the left side of the second precondition, such that it is no longer contained in the common segment of the rules.

Applying this transformation to the *while*-loop example, we get the following rules. Note that S is the only temporary variable in the rules for **while** and V_1 is the only temporary variable in the rules for **bool**.

$$\frac{B \triangleright [[[S]|D],S] \to [[[S]|D],Y] \quad factor_1(C,B) \triangleright [D,[[S],Y]] \to [D,S']}{\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od} \triangleright [D,S] \to [D,S']}$$

$$\frac{C;\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od} \triangleright [D,S] \to [D,S']}{factor_1(C,B) \triangleright [D,[[S],true]] \to [D,S']} \qquad \frac{}{factor_1(C,B) \triangleright [D,[[S],false]] \to [D,S]}$$

$$\frac{B \triangleright [D,S] \to [D,V_1] \quad test \triangleright [[[V_1]|D],[V_1]] \to [[[V_1]|D],R] \quad factor_2 \triangleright [D,[[V_1],R]] \to [D,V_2]}{\mathbf{bool}(B) \triangleright [D,S] \to [D,V_2]}$$

$$\frac{}{factor_2 \triangleright [D,[[V],false]] \to [D,\mathbf{type\_error}]} \qquad \frac{}{factor_2 \triangleright [D,[[V],true]] \to [D,V]}$$

$$\frac{}{test \triangleright [D,[V]] \to [D,is\_bool(V)]}$$

5.2.6. *Removing Variables First Defined in Preconditions*

Up to now, we did not restrict 2BIG rules such that instructions in the preconditions contain only variables defined in the left hand side of the conclusion.[5] We say that source and input state variables (see Def. 5.3) are **conclusion-defined**. If a variable is not conclusion-defined, then it is first defined in a precondition. All transformations so far only introduce preconditions with conclusion-defined variables in instructions. For the later transformation to TRS we need the property that all preconditions have only conclusion-defined variables in instructions. The following transformation converts all preconditions into the restricted form. A precondition with an instruction which contains variables first defined in a precondition is replaced by a precondition with a new instruction only containing conclusion-defined variables.

Let $\frac{c_1 \triangleright e_1 \to e_1' \ \ldots \ c_n \triangleright e_n \to e_n'}{c \triangleright e \to e'}$ be a rule. For all $i > 1$ we define: $\mathscr{M}_i = \{x : x \in \mathscr{V}(c_i)\ and\ x \notin \mathscr{V}(c,e)\}$. For every $\mathscr{M}_i \neq \emptyset$ a new instruction p_i is introduced and the rule is transformed into

$$\frac{c_1 \triangleright e_1 \to e_1' \ \ldots \ c_{i-1} \triangleright e_{i-1} \to e_{i-1}' \ p_i \triangleright e_{i-1}' \to e_i' \ c_{i+1} \triangleright e_{i+1} \to e_{i+1}' \ \ldots \ c_n \triangleright e_n \to e_n'}{c \triangleright e \to e'}$$

The new instructions are defined by the rules

$$\frac{c_i \triangleright e_i \to e_i'}{p_i \triangleright e_{i-1}' \to e_i'}$$

where ι_i is a new instruction symbol, $p_i = \iota_i(x_1,\ldots,x_k)$ and $\{x_1,\ldots,x_k\} = \mathscr{V}(c) \cap \mathscr{V}(c_i)$.

Recall that after allocation of temporary variables the term e_{i-1}' contains all non-source variables, which might be used in c_i, e_i.

[5] In [McK94] the form of rules has been much more restrictive: $\frac{c_1 \triangleright e_1 \to e_1' \ \ldots \ c_n \triangleright e_n \to e_n'}{a(p_1,\ldots,p_m) \triangleright e \to e'}$ where $\{c_1,\ldots,c_n\} \subseteq \{p_1,\ldots,p_m\}$, i.e., a command is executed by executing some of its arguments.

In the following example, the definition E of a procedure X is looked up in the state and then executed. The first defining occurrence of E is in the first precondition:

$$\frac{\mathbf{lookup}(X) \triangleright S \to E \quad E \triangleright S \to S'}{\mathbf{call}(X) \triangleright S \to S'}$$

After allocation of temporary variables we get the following rule. The first defining occurrence of E is still in the first precondition:

$$\frac{\mathbf{lookup}(X) \triangleright [[[S]|D], S] \to [[[S]|D], E] \quad E \triangleright [D, S] \to [D, S']}{\mathbf{call}(X) \triangleright [D, S] \to [D, S']}$$

Now the above transformation can be applied:

$$\frac{\mathbf{lookup}(X) \triangleright [[[S]|D], S] \to [[[S]|D], E] \quad exec \triangleright [[[S]|D], E] \to [D, S']}{\mathbf{call}(X) \triangleright [D, S] \to [D, S']}$$

$$\frac{E \triangleright [D, S] \to [D, S']}{exec \triangleright [[[S]|D], E] \to [D, S']}$$

Now E is an input state variable in the second rule and can thus occur in the instructions of its preconditions.

5.2.7. *Sequentialisation*

Next we will transform the rules such that the state on the right side of a precondition is equal to the state on the left side of the subsequent precondition. Furthermore the state on the right side of the last precondition is equal to the state on the right side of the conclusion.

More precisely, a rule of the form

$$\frac{c_1 \triangleright e_1 \to e_1' \quad \ldots \quad c_n \triangleright e_n \to e_n'}{c_0 \triangleright e_0 \to e_{n+1}}$$

is transformed into

$$\frac{c_1 \triangleright e_1 \to e_1' \quad p_1 \triangleright e_1' \to e_2 \quad \ldots \quad p_{n-1} \triangleright e_{n-1} \to e_n \quad c_n \triangleright e_n \to e_n' \quad p_n \triangleright e_n' \to e_{n+1}}{c_0 \triangleright e_0 \to e_{n+1}}$$

and we add the rules

$$p_1 \triangleright e_1' \to e_2 \quad \ldots \quad p_n \triangleright e_n' \to e_{n+1}$$

where for each rule of the form $p_i \triangleright e_i' \to e_{i+1}$, the instruction p_i has the form $\iota(x_1, \ldots, x_k)$, ι is a new instruction symbol and $\{x_1, \ldots, x_k\} = (\mathscr{V}(e_{i+1}) - \mathscr{V}(e_i')) \cap \mathscr{V}(c_0)$.

One might ask, why we do not add a precondition $p_0 \triangleright e_0 \to e_1$? But this case is taken care of by the next transformation, namely the conversion into term rewriting rules. Note that after the allocation of temporary variables, there should be no variables except for anonymous variables and source variables in $\mathscr{V}(e_{i+1}) - \mathscr{V}(e_i')$.

Applying this transformation to our *while*-loop example new instructions are introduced and the following rules are generated:

$$\frac{B \triangleright [[[S]|D], S] \to [[[S]|D], Y] \quad conv_1 \triangleright [[[S]|D], Y] \to [D, [[S], Y]] \quad factor_1(C, B) \triangleright [D, [[S], Y]] \to [D, S']}{\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od} \triangleright [D, S] \to [D, S']}$$

$$\frac{C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od} \triangleright [D, S] \to [D, S']}{factor_1(C, B) \triangleright [D, [[S], true]] \to [D, S']} \qquad \frac{}{factor_1(C, B) \triangleright [D, [[S], false]] \to [D, S]}$$

$$\frac{}{conv_1 \triangleright [[[S]|D], Y] \to [D, [[S], Y]]}$$

$$\frac{\begin{array}{l} B \triangleright [D, S] \to [D, V_1] \quad conv_2 \triangleright [D, V_1] \to [[[V_1]|D], [V_1]] \\ test \triangleright [[[V_1]|D], [V_1]] \to [[[V_1]|D], R] \quad conv_3 \triangleright [[[V_1]|D], R] \to [D, [[V_1], R]] \\ factor_2 \triangleright [D, [[V_1], R]] \to [D, V_2] \end{array}}{\mathbf{bool}(B) \triangleright [D, S] \to [D, V_2]}$$

$$\frac{}{factor_2 \triangleright [D, [[V], false]] \to [D, \mathbf{type\_error}]} \qquad \frac{}{factor_2 \triangleright [D, [[V], true]] \to [D, V]}$$

$$\frac{}{test \triangleright [D, [V]] \to [D, is\_bool(V)]} \qquad \frac{}{conv_2 \triangleright [D, V] \to [[[V]|D], [V]]} \qquad \frac{}{conv_3 \triangleright [[[V]|D], R] \to [D, [[V], R]]}$$

5.2.8. Conversion into Term Rewriting Systems

After sequentialisation, the instructions of each precondition can be proved in the state resulting from its preceding precondition. This property enables us to combine the instructions and generate term rewriting rules: Rules of the form $c \triangleright e \to e'$ are transformed into the rewrite rule $\langle (c;p), e\rangle \Rightarrow \langle p, e'\rangle$, where p is a new variable name, which will be bound to the program rest when the rule is applied. Rules of the form

$$\frac{c_1 \triangleright e_1 \to e'_1 \quad \ldots \quad c_n \triangleright e_n \to e'_n}{c \triangleright e \to e'}$$

are converted into

$$\langle (c;p), e\rangle \to \langle (c_1;\ldots;c_n;p), e_1\rangle$$

where p is a new variable name.

The term rewriting system generated for our *while*-loop example is:

$$\begin{array}{lcl}
\langle \textbf{while } B \textbf{ do } C \textbf{ od}; P, [D,S]\rangle & \Rightarrow & \langle B; conv_1; factor_1(C,B); P, [[[S]|D], S]\rangle \\
\langle conv_1; P, [[[S]|D], Y]\rangle & \Rightarrow & \langle P, [D, [[S], Y]]\rangle \\
\langle factor_1(C,B); P, [D, [[S], true]]\rangle & \Rightarrow & \langle C; \textbf{while } B \textbf{ do } C \textbf{ od}; P, [D,S]\rangle \\
\langle factor_1(C,B); P, [D, [[S], false]]\rangle & \Rightarrow & \langle P, [D,S]\rangle \\
\langle \textbf{bool}(B); P, [D,S]\rangle & \Rightarrow & \langle B; conv_2; test; conv_3; factor_2; P, [D,S]\rangle \\
\langle conv_2; P, [D,V]\rangle & \Rightarrow & \langle P, [[[V]|D], [V]]\rangle \\
\langle conv_3; P, [[[V]|D], R]\rangle & \Rightarrow & \langle P, [D, [[V], R]]\rangle \\
\langle factor_2; P, [D, [[V], true]]\rangle & \Rightarrow & \langle P, [D, V]\rangle \\
\langle factor_2; P, [D, [[V], false]]\rangle & \Rightarrow & \langle P, [D, \textbf{type_error}]\rangle \\
\langle test; P, [D, [V]]\rangle & \Rightarrow & \langle P, [D, is_bool(V)]\rangle
\end{array}$$

The term rewriting systems produced by conversion of 2BIG rules are linear and orthogonal, because the left hand sides of the rules are non-overlapping. By Theorem 5.4 we conclude that these term rewriting systems are confluent.

5.2.9. Pass Separation

The term **staging transformation** has been introduced in [JøS86] for a class of transformations including partial evaluation and pass separation. Let p be a program, x and y the static and dynamic inputs to this program and $\overline{x}$ the statically known value of x, then **partial evaluation** of p with respect to $\overline{x}$ yields a residual program $p_{\overline{x}}$, such that $p_{\overline{x}}(y) = p(\overline{x}, y)$. In contrast, **pass separation** transforms the program p into two programs p_1 and p_2 such that $p_2(p_1(x), y) = p(x, y)$. What is important about this equation is that here p_1 produces some intermediate data, which are input to p_2. Note that there are trivial solutions to the above equations, namely $p_{\overline{x}} = \lambda y.(\lambda x.p(x, y)\ \overline{x})$ respectively $p_1 = id$ and $p_2 = p$. The goal is to move computations from p to $p_{\overline{x}}$ respectively from p_2 to p_1.

As is well known, partial evaluation can be used to generate compilers in various ways according to the Futamura Projections [JGS93, Fut71]. The drawback of this approach is that the generated code is in the interpreter's language and requires its evaluation mechanism. Thus, partial evaluation will not devise a target language suitable for the source language or invent new runtime data structures. When it comes to the generation of compiler/executor pairs, pass separation provides an immediate solution. We pass separate the interpreter *interp* into an executor *exec* and a compiler *comp*, such that: $interp(prog, data) = exec(comp(prog), data)$. Despite this potential for compiler generation there is only little work on pass separation. Actually we are only aware of the intuitive presentation in [JøS86] and the provably correct pass separation transformations for term rewriting systems [Han94] and evolving algebras [Die97b].

5.2.10. Hannan's Pass Separation Transformation for Abstract Interpreters

In this section we only cite the relevant definitions of the key paper [Han94] with minimal explanation. The interested reader will find more explanations, examples and a correctness proof in Hannan's article.

Definition 5.8 (Term Complexity). The complexity of a term t is defined by structural induction: $\mathscr{C}(\iota) = 1$ for a constant ι; $\mathscr{C}(X) = 1$ for a variable X; and $\mathscr{C}(\iota(t_1, \ldots, t_n)) = \mathscr{C}(t_1) + \cdots + \mathscr{C}(t_n) + 1$. From this we derive a partial order on terms: $t_1 \sqsubset t_2$ iff $\mathscr{C}(t_1) < \mathscr{C}(t_2)$.

Definition 5.9 (Abstract Interpreter (Def. 4.1 in [Han94])**).** (Σ, R) is an **abstract interpreter** iff

- (Σ, R) is a linear term rewriting system and nop, $\langle .,. \rangle$, $';' \in \Sigma$
- every rule in R is of the form: $\langle \iota(X_1,\ldots,X_k); C, e\rangle \Rightarrow \langle c'_1;\ldots;c'_m; C, e'\rangle$ where $X_1,\ldots,X_k$ and C are variables and $c'_1,\ldots,c'_m$ and e, e' are terms which do not contain C.

The term rewriting system generated in Section 5.2.8 for our running example is an abstract interpreter.

Definition 5.10 (Instruction Defining Rules (Def. 6.1 in [Han94])**).** Let (Σ, R) be an **abstract interpreter**. For a constant $\iota \in \Sigma$, $R|_{\iota,k}$ is the set containing those rules of the form $\langle \iota(X_1,\ldots,X_k); C, e\rangle \Rightarrow \langle p', e'\rangle$ for some e, e' and p'.

Definition 5.11 (Atomic Program). A term of the form $\iota(s_1,\ldots,s_n)$ is called an **atomic program**, where ι is a constant and none of the terms s_i contains one of the constants nop, $\langle .,. \rangle$ or $';'$.

Definition 5.12 (Common Suffix (Def. 6.2 in [Han94])**).** Let (Σ, R) be an **abstract interpreter**. If the rule $\langle \iota(X_1,\ldots,X_k); C, e\rangle \Rightarrow \langle p', e'\rangle$ is in $R|_{\iota,k}$, then $\mathsf{suffix}(R|_{\iota,k})$ is the term $b' = b_1;\ldots;b_n$ (for atomic programs b_i) such that

1. $\mathscr{V}(b') \subseteq \{X_1,\ldots,X_k\}$
2. $b_i \sqsubset \iota(X_1,\ldots,X_k)$, for all $1 \leqslant i \leqslant n$
3. for every rule $r \in R|_{\iota,k}$ there exist atomic programs $a_1,\ldots,a_m$ and terms e, e' such that r is of the form $\langle \iota(X_1,\ldots,X_k); C, e\rangle \Rightarrow \langle a_1;\ldots;a_m; b'; C, e'\rangle$,
4. for no term $b_0; b_1;\ldots;b_n$ do the previous three properties hold.

(1) Ensures that b' can be constructed at compile-time (see our rationale in Section 5.2.1), (2) ensures that the compiler will be normalising, i.e., compilation will terminate, (3) is needed to show confluence of the generated compiler (by Theorem 5.4) and (4) enforces as much reduction as possible at compile-time. Pass separation detects such parts of the term rewriting rule which can be rewritten independently of the state, i.e., at compile-time.

To construct the compiler one divides the instructions (atomic programs) on the right hand side $p; C$ of an abstract interpreter rule $\langle \iota(X_1,\ldots,X_k); C, e\rangle \Rightarrow \langle p; C, e'\rangle \in R$ for $\iota(X_1,\ldots,X_k)$ into two parts: $a_1;\ldots;a_m$ is characteristic for the rule r and $b'; C$ is common to all rules in $R|_{\iota,k}$. Then a compiler rule is generated which translates $\iota(X_1,\ldots,X_k)$ into the program $\kappa(X_1,\ldots,X_k); b'$ where κ is a new constant. The 'execution' of the characteristic instructions and the change of the state e to e' is done by the generated executor rule for κ.

Definition 5.13 (Pass Separation (Def. 6.3 in [Han94])**).** Let (Σ, R) be an **abstract interpreter**.

1. (Σ_c, R_c) is the smallest rewrite system such that
   if $\langle \iota(X_1,\ldots,X_k); C, e\rangle \Rightarrow \langle a_1;\ldots;a_m; b'; C, e'\rangle \in R$ and $b' = \mathsf{suffix}(R|_{\iota,k})$
   then $\iota(X_1,\ldots,X_k) \Rightarrow_c \kappa(X_1,\ldots,X_k); b' \in R_c$, where κ is a new constant.
2. (Σ_x, R_x) is the smallest rewrite system such that if
   $\langle \iota(X_1,\ldots,X_k); C, e\rangle \Rightarrow \langle a_1;\ldots;a_m; b'; C, e'\rangle \in R|_{\iota,k}$ and $b' = \mathsf{suffix}(R|_{\iota,k})$, $\iota(X_1,\ldots,X_k) \Rightarrow_c \kappa(X_1,\ldots,X_k); b' \in R_c$, $a_1;\ldots;a_m \overset{*}{\Rightarrow}_{R_c} a_c$, $e \overset{*}{\Rightarrow}_{R_c} e_c$ and $e' \overset{*}{\Rightarrow}_{R_c} e'_c$ then $\langle \kappa(X_1,\ldots,X_k); C, e_c\rangle \Rightarrow \langle a_c; C, e'_c\rangle \in R_x$

Pass separation converts a set of rules R into two sets R_c and R_x, such that if $\langle c, e\rangle \overset{*}{\Rightarrow}_R \langle c', e'\rangle$ then $c \overset{\dagger}{\Rightarrow}_{R_c} \tilde{c}$, $e \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e}$, $c' \overset{\dagger}{\Rightarrow}_{R_c} \tilde{c}'$, $e' \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e}'$ and $\langle \tilde{c}, \tilde{e}\rangle \overset{*}{\Rightarrow}_{R_x} \langle \tilde{c}', \tilde{e}'\rangle$.

Note that also the state is compiled, because there might be instruction sequences stored in the state. This occurs for example in the semantics of higher order languages. Compilation with R_c produces normal forms ($\overset{\dagger}{\Rightarrow}_{R_c}$).

In the terminology of John Hannan, the rules in R define an abstract interpreter, the rules in R_c a compiler and the rules in R_x an abstract executor, or in our terminology, an abstract machine. The rules in R_x belong to a special class of rewrite rules. Their left sides will only match the whole term (enclosed in $\langle\ldots\rangle$), i.e., they never apply to sub-terms of that term, because we do not allow $\langle\ldots\rangle$ to occur in a term. As a result they can be implemented more efficiently than ordinary rewrite rules.

Pass separating the TRS for our example results in the following compiler rules:

$$\begin{array}{lll} \textbf{while } B \textbf{ do } C \textbf{ od} & \Rightarrow & \overline{while}(B,C) \\ \textbf{bool}(B) & \Rightarrow & \overline{bool}(B);B;conv_2;test;conv_3;factor_2 \end{array}$$

$$conv_2 \Rightarrow \overline{conv}_2 \quad conv_3 \Rightarrow \overline{conv}_3 \quad factor_2 \Rightarrow \overline{factor}_2 \quad test \Rightarrow \overline{test}$$

And the following executor rules are generated:

$$\begin{array}{lll} \langle \overline{while}(B,C);P,[D,S]\rangle & \Rightarrow & \langle B;\overline{conv};\overline{factor}_1(C,B);P,[[[S]|D],S]\rangle \\ \langle \overline{conv};P,[[[S]|D],Y]\rangle & \Rightarrow & \langle P,[D,[[S],Y]]\rangle \\ \langle \overline{factor}_1(C,B);P,[D,[[S],true]]\rangle & \Rightarrow & \langle C;\overline{while}(B,C);P,[D,S]\rangle \\ \langle \overline{factor}_1(C,B);P,[D,[[S],false]]\rangle & \Rightarrow & \langle P,[D,S]\rangle \\ \langle \overline{bool}(B);P,[D,S]\rangle & \Rightarrow & \langle P,[D,S]\rangle \\ \langle \overline{conv}_2;P,[D,V]\rangle & \Rightarrow & \langle P,[[[V]|D],[V]]\rangle \\ \langle \overline{conv}_3;P,[[[V]|D],R]\rangle & \Rightarrow & \langle P,[D,[[V],R]]\rangle \\ \langle \overline{factor}_2;P,[D,[[V],true]]\rangle & \Rightarrow & \langle P,[D,V]\rangle \\ \langle \overline{factor}_2;P,[D,[[V],false]]\rangle & \Rightarrow & \langle P,[D,\textbf{type\_error}]\rangle \\ \langle \overline{test};P,[D,[V]]\rangle & \Rightarrow & \langle P,[D,is\_bool(V)]\rangle \end{array}$$

In this example we didn't get a compilation rule, which compiled the *while*-loop into a sequence of less complex instructions. Instead it is translated into the instruction $\overline{while}$, which performs the translation into $B;\overline{conv};\overline{factor}(C,B)$ at runtime. If we look at the definition of the pass separation transformation, we find that the term $B;\overline{conv};\overline{factor}(C,B)$ was a candidate for b', but $\overline{factor}(C,B)$ is as complex regarding number and structure of the arguments as the original **while** instruction ($\overline{factor}(C,B) \not\sqsubset$ **while** B **do** C **od**). Hannan only allows compiler rules which translate an instruction into a sequence of less complex instructions.[6] Using this restriction he can prove that the generated compiler is strongly normalising, i.e., terminates for all source language programs.

The compiler rule for **bool** produces a sequence of less complex machine instructions: $\textbf{bool}(B) \Rightarrow \overline{bool}(B);B;conv_2;test;conv_3;factor_2$ where $B \sqsubset \textbf{bool}(B), conv_2 \sqsubset \textbf{bool}(B)$, etc.

5.3. Optimisations of Compiler and Abstract Machine

After pass separation, the resulting compiler and abstract machine can be further optimised. One important goal of these optimisations is to reduce the number of instructions of the generated abstract machine.

5.3.1. Self-Application of Compiler Rules

After the right hand sides of each compiler rule have been compiled using the original compiler rules, we can remove all compiler rules for non-source language instructions, five for those instructions introduced by the transformations.

For example the 5 compiler rules for **bool** can be reduced to one rule

$$\textbf{bool}(B) \Longrightarrow \overline{bool}(B);B;\overline{conv}_2;\overline{test};\overline{conv}_3;\overline{factor}_2$$

5.3.2. Remove Unused Arguments

Pass separation sometimes introduces instructions with arguments which do not occur on the right side of any of the abstract machine rules for that instruction. Clearly such arguments can be removed. Thus we have to detect which arguments are not used, and then we have to remove those arguments in all occurrences of the instruction in the compiler as well as the abstract machine rules.

In our example the argument of the instruction $\overline{bool}(B)$ is not used on the right hand side of the abstract machine rule for that instruction. Thus we can simplify the compiler rule to $\textbf{bool}(B) \Longrightarrow \overline{bool};B;\overline{conv}_2;\overline{test};\overline{conv}_3;\overline{factor}_2$ and the abstract machine rule to $\langle \overline{bool};P,[D,S]\rangle \Longrightarrow \langle P,[D,S]\rangle$

Actually this instruction does not even change the state, so it could be completely removed.

[6] Restriction 2 of Definition 5.12.

5.3.3. Factorize Abstract Machine Rules

Sometimes several rules for two instructions are identical except for the instruction names. In this case, we can introduce a new instruction defined by the common rules. In our examples we could factorise rules dealing with error handling. Often the error handling was identical for several instructions.

Here is a simple example:

$\langle hd; C, [H|T]\rangle \Longrightarrow \langle C, H\rangle \qquad \langle hd; C, error\rangle \Longrightarrow \langle C, error\rangle$
$\langle tl; C, [H|T]\rangle \Longrightarrow \langle C, T\rangle \qquad \langle tl; C, error\rangle \Longrightarrow \langle C, error\rangle$

The error case can be factored out:

$\langle fact(hd); C, [H|T]\rangle \Longrightarrow \langle C, H\rangle \qquad \langle fact(tl); C, [H|T]\rangle \Longrightarrow \langle C, T\rangle$
$\langle fact(X); C, error\rangle \Longrightarrow \langle C, error\rangle$

In general the factorisation transformation for abstract machine rules works as follows:

Let R be the set of abstract machine rules. Let k_1 and k_2 be two instructions with the same number of arguments. Let $R_1 \subseteq R$ be the set of all rules for k_1 and $R_2 \subseteq R$ be the set of all rules for k_2. The intersection of two rule sets modulo instruction names is defined as:

$$R_1 \boxed{\cap} R_2 = \{ \quad [x_1, \ldots, x_n, c_1, e_1, s_1] \mid \langle k_1(x_1, \ldots, x_n); c_1,\ e_1\rangle \Rightarrow s_1 \in R_1 \text{ and } \langle k_2(y_1, \ldots, y_n); c_2,\ e_2\rangle \Rightarrow s_2 \in R_2 \text{ and exists a variable renaming } \theta \text{ such that } [x_1, \ldots, x_n, c_1, e_1, s_1] = \theta([y_1, \ldots, y_n, c_2, e_2, s_2])\}$$

The set of rules in R_1 which have no counterpart in R_2 is:

$$R_1 \boxed{-} R_2 = \{ \quad r_1 : r_1 = \langle k_1(x_1, \ldots, x_n); c_1,\ e_1\rangle \Rightarrow s_1 \in R_1 \text{ and there exists no } \langle k_2(y_1, \ldots, y_n); c_2,\ e_2\rangle \Rightarrow s_2 \in R_2 \text{ and no variable renaming } \theta \text{ such that } [x_1, \ldots, x_n, c_1, e_1, s_1] = \theta([y_1, \ldots, y_n, c_2, e_2, s_2])\}$$

Let ι be a new instruction symbol and x be a new variable name. We form a new set of rules $\widehat{R}$ to replace R_1 and R_2: For all $[x_1, \ldots, x_n, c, e, s] \in R_1 \boxed{\cap} R_2$ we have $\langle \iota(x, x_1, \ldots, x_n); c,\ e\rangle \Rightarrow s \in \widehat{R}$. Now we have to deal with those rules which are different for k_1 and k_2. For all $\langle k_1(x_1, \ldots, x_n); c,\ e\rangle \Rightarrow s \in R_1 \boxed{-} R_2$ we have $\langle \iota(k_1, x_1, \ldots, x_n); c,\ e\rangle \Rightarrow s \in \widehat{R}$. And similarily: For all $\langle k_2(x_1, \ldots, x_n); c,\ e\rangle \Rightarrow s \in R_2 \boxed{-} R_1$ we have $\langle \iota(k_2, x_1, \ldots, x_n); c,\ e\rangle \Rightarrow s \in \widehat{R}$. Finally we replace the rules R_1 and R_2 in R by $\widehat{R}$.

5.3.4. Combining Instructions

Transformation of side conditions and sequentialisation introduce instructions which are defined by a single rule. In the compiler rules, we can detect sequences of such instructions and combine them into a single instruction. This instruction has the combined effect of the underlying instructions, but can be executed in one step. This reduces the interpretation overhead, pattern matching, and the construction of intermediate data structures.

In what follows, let $c \Longrightarrow i_1; \ldots; i_n \in R_c$ be a compiler rule, $i_j; \ldots; i_{j+k}$ be the longest sequence ($k > 1$) of instructions in the above compiler rule such that each instruction is defined by a single abstract machine rule and let the rule defining the first instruction in the sequence be $\langle i_j; C, e_j\rangle \Longrightarrow \langle c'; C, e'_j\rangle \in R_x$.

In the following we mean by pure rewriting $\overset{*}{\hookrightarrow}_{R_x}$ that function names are treated as constructors and thus function calls are not evaluated. If we get that $\langle i_j; \ldots; i_{j+k}, e_j\rangle \overset{*}{\hookrightarrow}_{R_x} \langle nop, e'\rangle$ then we define a new instruction.[7] Let ι be a new instruction symbol and $\{x_1, \ldots, x_p\} = \bigcup_{m=j}^{j+k} \{a_1, \ldots, a_{q_m} : i_m = \iota_m(a_1, \ldots, a_{q_m})\}$

The compiler rule is replaced by $c \Longrightarrow i_1; \ldots; i_{j-1}; \iota(x_1, \ldots, x_p); i_{j+k+1}; \ldots; i_n$ and the following abstract machine rule is added: $\langle \iota(x_1, \ldots, x_p); C, e_j\rangle \Longrightarrow \langle C; e'\rangle$

As an example look at the compiler rule for **bool**(B). The sequence $\overline{conv}_2; \overline{test}; \overline{conv}_3;$ is replaced by a new instruction $\overline{comb}$: **bool**(B) $\Longrightarrow \overline{bool}; B; \overline{comb}; \overline{factor}_2$ The new instruction replaces the three original ones: $\langle \overline{comb}; P, [D, V]\rangle \Longrightarrow \langle P, [D, [[V], \overline{is\_bool}(V)]]\rangle$

A similar transformation replaces sequences of instructions in the right hand side of an abstract machine rule by a combined instruction.

[7] Otherwise, we try a shorter sequence, i.e., if $k > 2$, let $k = k - 1$ and try again.

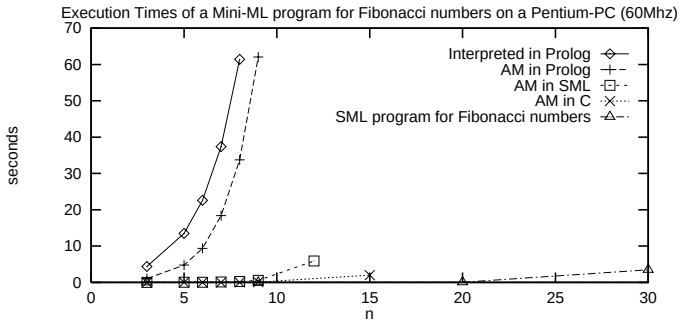


Fig. 3. Mini-ML: Fibonacci numbers.

5.3.5. Remove Redundant Rules

Due to the automatic generation of instructions, there are often instructions, which are defined by the same rules except for the instruction name, e.g., the pairwise factorisation of abstract machine rules leads to redundant rules, if more than two instructions have common rules. In all these cases, we can replace instructions which are equal modulo instruction names by one instruction. Then we can remove the rules of the replaced instructions from the abstract machine.

These optimisations greatly reduce the number of compiler and abstract machine rules, e.g. for the action notation specification by self-application the number of compiler rules was reduced from 216 to 43. Using the other optimisations we obtained 181 instead of 276 abstract machine rules.

6. Performance Evaluation

The main focus of our work has been feasibility and correctness. So far efficiency has not been the main goal of our work. Nevertheless to compare our approach to other approaches in semantics-directed compiler generation, we provide some benchmarks and performance figures.

The following benchmarks were run: A program called *loop* which counts from 1 to *n*, a program *fib* which computes the Fibonacci number of *n* and program *primes* which computes the first *n* prime numbers using the algorithm known as 'Sieve of Eratosthenes'.

We did benchmarks with three different languages: SIMP, Mini-ML and Action Notation. The constructs of these languages include recursive functions and procedures, higher-order functions, local and global variables, assignments, conditionals and loops. Only for SIMP we tested all three programs, as the other languages did not provide arrays which were used in the *primes* program. For SIMP we used an an iterative *fib* program; for Mini-ML only *loop* and a recursive *fib* program were tested, because Mini-ML does not provide arrays which we need for the *primes* program. Mini-Δ is an imperative language with procedures and functions and its semantics is given in Action Notation. The action terms produced by expanding Mini-Δ programs of *loop* and a recursive *fib* were compiled and executed with the generated compiler and abstract machine. For the actual performance figures and graphs see [Die96].

SIMP. Both the generated abstract machine in SML and the abstract machine in C are considerably faster than the interpretation in Prolog. In Prolog and SML we soon ran into memory management problems, whereas in C also for greater values of *n* garbage collection did not degrade performance. For the *primes* program we ran out of memory when interpreting the 2BIG rules or the generated abstract machine in Prolog, but the generated abstract machine program in C worked fine, and was 2 orders of magnitude slower than a C version of the *primes* program.

Mini-ML**.** Executing a compiled example program on the generated abstract machine for Mini-ML [Kah87, Des86] in C is 2 orders of magnitude faster than interpreting the 2BIG rules in Prolog. On the other hand it is 2 orders of magnitude slower than an equivalent SML program compiled with the SML of New Jersey Compiler (see also Fig. 3). Actually, the generated abstract machine is very close to the CAM, an abstract machine used for efficient implementations of ML. For the CAM the state is a stack of environments; in our

generated abstract machine the state has the form $[D,[R,E]]$, where R are redirections and D corresponds to the rest of the stack in the CAM and E to the topmost element of the stack. Taking this into account, there is a one-to-one correspondence of about half of the CAM instructions `car`, `cdr`, `cur`, `swap`, `app` to instructions of the generated machine. For the `push` instruction the system produces 3 and for the `cons` instruction 2 more specialised instructions. The major deviation of the generated instructions is the explicit handling of redirections. In the specifications of the CAM these are hidden in the meta-language (cyclic bindings). A complete comparison is presented in [Die00].

Action Notation. Mini-Δ [MoW94] is an imperative language with procedures and functions. It allows both call-by-value and call-by-reference parameter passing. Furthermore functions can be passed as parameters. For the performance tests a benchmark program in the language Mini-Δ was first translated into an action term using an Action Semantics specification of Mini-Δ. This action term was then executed by interpreting the 2BIG rules for Action Semantics or by the generated abstract machine for Action Semantics in C. The abstract machine in C was at least 2 orders of magnitude faster than interpreting the rules. The abstract machine and the compiler were generated from 100 2BIG rules defining the semantics of 39 action notation constructs including the control, functional, declarative and imperative facet. We did not deal with the communicative facet, nondeterminism or the interleaving of actions. More details on this and further experiments with the Action Notation specification are presented in [Die99].

7. Correctness

Most of the transformations in our system convert a set ϕ of relational inductive rules into another set ϕ' of relational inductive rules. Let T be such a transformation and R be a relation on terms in $T_\Sigma(X)$. In [Die96] we prove claims of the form:

Theorem: For all $(x,y) \in R$ we have: $x \in \mathscr{I}(\overline{\phi}) \Leftrightarrow y \in \mathscr{I}(\overline{\phi'})$

Actually to prove such a theorem we prove two lemmata. The theorem follows immediately from these.

Completeness-Lemma: For all $(x,y) \in R$ we have: $x \in \mathscr{I}(\overline{\phi}) \Rightarrow y \in \mathscr{I}(\overline{\phi'})$

Soundness-Lemma: For all $(x,y) \in R$ we have: $x \in \mathscr{I}(\overline{\phi}) \Leftarrow y \in \mathscr{I}(\overline{\phi'})$

Our correctness proofs are based on proof trees and use Theorem 2.4 in [Die96], which says that for every element in the meaning $(\mathscr{I}(\overline{\phi'}))$ of a set of 2BIG rules ϕ there exists a proof tree and vice versa. To prove completeness we arbitrarily choose $x \in \mathscr{I}(\overline{\phi})$. Then Theorem 2.4 implies that there exists a $\overline{\phi}$-tree $\frac{P\,T_{\overline{\phi}}(b_1)\ldots P\,T_{\overline{\phi}}(b_n)}{x}$ for x. Next we use the definitions of the transformation T and the relation R to construct a $\overline{\phi'}$-tree $\frac{P\,T_{\overline{\phi'}}(b'_1)\ldots P\,T_{\overline{\phi'}}(b'_m)}{y}$ for y. We use induction on the depth of the $\overline{\phi}$-tree. Now Theorem 2.4 implies $y \in \mathscr{I}(\overline{\phi'})$. In a similar way we prove soundness.

Correctness of Core System

Let T_F be the factorisation transformation, T_P the pass separation, T_{TRS} the conversion into TRS, T_S the sequentialisation transformation, T_R the removal of variables and T_A the allocation of temporary variables. These transformations form the core system. Let $T^0(\phi) = \phi$ and $T^i(\phi) = T(T^{i-1}(\phi))$ be the iterative application of a transformation T. By the theorems proven in [Die96] follows:

Theorem 7.1. Let ϕ be a set of 2BIG rules, $\phi' = T_F^m(\phi)$ such that $T_F^m(\phi) = T_F^{m+1}(\phi)$ and $(R_x, R_c) = T_P(T_{TRS}(T_S(T_R(T_A(\phi')))))$. Then

1. if $c \triangleright e \to e' \in \mathscr{I}(\overline{\phi})$ then $c \overset{\dagger}{\Rightarrow}_{R_c} \tilde{c}$, $e \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e}$, $e' \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e'}$
   and $\langle \tilde{c}, [[],\tilde{e}]\rangle \overset{*}{\Rightarrow}_{R_x} \langle nop, [[],\tilde{e'}]\rangle$
2. if $c \overset{\dagger}{\Rightarrow}_{R_c} \tilde{c}$, $e \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e}$ and $\langle \tilde{c}, [[],\tilde{e}]\rangle \overset{*}{\Rightarrow}_{R_x} \langle nop, [[],\tilde{e'}]\rangle$ then $c \triangleright e \to e' \in \mathscr{I}(\overline{\phi})$ and $e' \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e'}$

Intuitively this theorem states that if the execution of a program c in a start state e according to the semantics specified by the 2BIG rules ϕ yields final state e' (short: $c \triangleright e \to e' \in \mathscr{I}(\overline{\phi})$), then the execution of the compiled program $\tilde{c}$ in a corresponding start state yields a corresponding final state. This correspondence of states in the semantics and the abstract machine is expressed in terms of compilation. This is necessary

because we allow instructions to occur in states, e.g., the state might contain an environment mapping names to function abstractions as in the case of the Mini-ML specification. These function abstractions contain instructions and these instructions have to be compiled into the instructions of the abstract machine.

8. Concluding Remarks

So far the derivation of abstract machines has not been assisted by automated tools. In this article, we presented a system that automatically generates a compiler and abstract machine from a 2BIG specification of a programming language. The compiler computes only on program structures, whereas the abstract machine computes primarily on run-time structures.

In [Die96] we give the transformations used in our system, its correctness proof and discuss the results of applying our system to the 2BIG specifications of two toy languages, namely SIMP and Mini-ML , and a specification of Action Semantics. For Mini-ML , we get an abstract machine which is very close to the CAM, an abstract machine used for efficient implementations of ML. We also used the system to generate a compiler and abstract machine for action notation. These have been used as a backend of an action semantics-based compiler generator. Our results are backed by a running implementation tested with non-trivial examples as well as a correctness proof of the core system.

The term rewriting rules generated by our system, which define the compiler and abstract machine, can also be used as a basis for handwritten compilers and abstract machines, much in the way efficient compilers and abstract machines have been written starting from the term rewriting systems given in [CCM85]. This is in contrast to the traditional partial evaluation approaches to compiler generation. First, by simply composing semantics equations and a partial evaluator no language-specific compiler is generated. Second, by self-application of the partial evaluator, the generated compilers inherit much structure from the underlying partial evaluator and the analysis and code generation parts are intertwined, making it difficult to use these residual programs as starting points for handwriting compilers.

9. Future Work

Admittedly the transformations introduce many abstract machine instructions. The number of conversion instructions introduced by sequentialisation should be reduced by replacing similar instructions by one more general instruction. Furthermore there are instructions which do nothing besides pattern matching, i.e., test whether the current state has a required form, and it might be safe to remove them in case we know that the state will always have the required form. Thus these and other optimisations have to be investigated further. Such optimisations are likely to be based on global analyses. Detection of single-threadedness and compile-time garbage collection have been studied in the functional world and might help to generate more efficient implementations of abstract machines. As our system is the first running implementation of a semantics-directed compiler generator which uses pass separation as its key transformation and our performance results are encouraging, we feel that the development of pass separation transformations for other meta-languages is a challenging and worthwhile future research goal.

Up to now, our system has only been used to generate compilers from natural semantics specifications of programming languages. These specifications can be regarded as interpreters. As one can also specify other functionalities like static semantics checking or program specialization algorithms using natural semantics rules, it is possible to pass separate those specifications.

References

[Acz77] Aczel, P.: An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*. North-Holland, 1977.

[ASU86] Aho, A. V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AhU72] Aho, A. V. and Ullman, J. D.: *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, 1972.

[Aït91] Aït-Kaci, H.: *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[Ast91] Astesiano, E.: Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, IFIP, 1991.

[Ber91] Berry, D.: *Generating Program Animators from Programming Language Semantics*. Technical Report ECS-LFCS-91-163, University of Edinburgh, 1991.

[BFH94] Böschen, C., Fecht, C., Hense, A. V., and Wilhelm, R.: An abstract machine for an object-oriented language with top-level classes. Technical Report FB14 - No. A 02/94, Computer Science Department, University of Saarbrücken, 1994.

[Car84] Cardelli, L.: Compiling a functional language. In *International Symposium on LISP and Functional Programming*. Austin, Texas, ACM, 1984.

[CCM85] Cousineau, G., Curien, P.-L. and Mauny, M.: The categorical abstract machine. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture FPCA'85*, volume LNCS 201. Springer-Verlag, 1985.

[DaM82] Damas, L. and Milner, R.: Principal type-schemes for functional languages. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages POPL'82*. Alberquerque, New Mexico, ACM, 1982.

[daS90] da Silva, F. Q. B.: Towards a formal framework for evaluation of operational semantics. Technical Report ECS-LFCS-90-126, Edinburgh University, 1990.

[daS92] da Silva, F. Q. B.: *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics*. PhD thesis, University of Edinburgh, 1992.

[Des84] Despeyroux, T.: Executable specification of static semantics. In G. Kahn, D. B. MacQueen and G. Plotkin, editors, *Proceedings of International Symposium on Semantics of Data Types*, volume LNCS 173. Springer-Verlag, 1984.

[Des86] Despeyroux, J.: Proof of translation in natural semantics. In *Proceedings of the Symposium on Logic in Computer Science LICS '86*. IEEE Computer Society, 1986.

[Die96] Diehl, S.: *Semantics-Directed Generation of Compilers and Abstract Machines*. Ph.D. thesis, University Saarbrücken, Germany, 1996. http://www.cs.uni-sb.de/~diehl/phd.html.

[Die97a] Diehl, S.: An experiment in abstract machine design. *Software – Practice and Experience*, 27(1): 49–62, 1997.

[Die97b] Diehl, S.: Transformations of evolving algebras. In *Proceedings of VIII International Conference on Logic and Computer Science LIRA'97*, pages 43–50. Novi Sad, Yugoslavia, 1997.

[Die99] Diehl, S.: Bootstrapped semantics-directed compiler generation. In P. D. Mosses and D. A. Watt, editors, *AS'99, Proceedings of Second International Workshop on Action Semantics*, Amsterdam, The Netherlands, number NS-99-3 in Notes Series, BRICS, Department of Computer Science, University of Aarhus, May 1999. Available from `http://www.brics.dk/NS/99/3`.

[Die00] Diehl, S.: A generative methodology for the design of abstract machines. *Science of Computer Programming*, Vol. 38, pages 125–142, 2000.

[DHS99] Diehl, S. Hartel, P. and Sestoft, P.: Abstract machines for programming language implementation. *Future Generation Computer Systems*, Vol. 16(7), Elsevier, pages 739–751, 1999.

[Fut71] Futamura, Y.: Partial evaluation of computation process: An approach to a compiler–compiler. *Systems, Computers, Controls*, 2(5): 45–50, 1971.

[Han94] Hannan, J.: Operational semantics-directed compilers and machine architectures. *ACM Transactions on Programming Languages and Systems TOPLAS*, 16(4): 1215–1247, 1994.

[Hue80] Huet, G.: Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4): 797–821, 1980.

[JSM92] Jaffar, J., Stuckey, P. J., Michaylov, S. and Yap, R. H. C.: An abstract machine for CLP($\mathscr{R}$). In *PLDI'92*, San Francisco. Sigplan Notices, 1992.

[JGS93] Jones, N. D., Gomard, C. K. and Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[Joh84] Johnson, T.: Efficient compilation of lazy evaluation. In *CC'84*. Sigplan Notices 19(6), 1984.

[Jou95] Jouannaud, J.-P.: Introduction to rewriting. In H. Comon and J.-P. Jouannaud, editors, *Term Rewriting*, volume LNCS 909. Springer-Verlag, 1995.

[JøS86] Jørring, U. and Scherlis, W. L.: Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, pp. 86–96. ACM Press, 1986.

[Kah87] Kahn, G.: Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, volume LNCS 247, pp. 22–39. Springer-Verlag, 1987.

[Kur86] Kursawe, P.: How to invent a Prolog machine. In *Proceedings of Third International Conference on Logic Programming*, pp. 134–148. LNCS 225, Springer-Verlag, 1986.

[Lan64] Landin, P. J.: The mechanical evaluation of expressions. *Computer Journal*, 6(4): 308–320, 1964.

[Lee89] Lee, P.: *Realistic Compiler Generation*. MIT Press, 1989.

[Lem92] Lemone, K. A.: *Design of Compilers: Techniques of Programming Language Translation*. CRC Press, 1992.

[LeP81] Lewis, H. R. and Papadimitriou, C. H.: *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[Llo87] Lloyd, J. W.: *Foundations of Logic Programming*, 2nd edn. Springer-Verlag, 1987.

[McK94] McKeever, S.: A framework for generating compilers from natural semantics specifications. In P. D. Mosses, editor, *Proceedings of the 1st Workshop on Action Semantics*, BRICS-NS-94-1. University of Aarhus, Denmark, 1994.

[MSS95] Mehl, M., Scheidhauer, R. and Schulte, C.: An abstract machine for OZ. In M. Hermenegildo and S. D. Swierstra, editors, *7th International Symposium, PLILP'95*, volume LNCS 982. Springer-Verlag, 1995.

[MTH90] Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*. MIT Press, 1990.

[Mos92] Mosses, P. D.: *Action Semantics*. Cambridge University Press, 1992.

[Mou93] Moura, H.: *Action Notation Transformations*. PhD thesis, University of Glasgow, 1993.

[MoW94] Moura, H. and Watt, D. A.: Action transformations in the ACTRESS compiler generator. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of 6th International Workshop on Languages and Compilers for Parallel Computing CC'94*, volume LNCS 768. Springer-Verlag, 1994.

[Müc92] Mück, A.: Camel: An extension of the categorical abstract machine to compile functional logic programs. In *PLILP'92*, volume LNCS 631. Springer-Verlag, 1992.

[NiN86] Nielson, F. and Nielson, H. R.: Code generation from two-level denotational meta-languages. In H. Ganzinger, N. D. Jones, editors, *Programs as Data Objects*, volume LNCS 217. Springer-Verlag, 1986.

[NiN92] Nielson, H. R. and Nielson, F.: *Semantics with Applications – A Formal Introduction*. John Wiley, 1992.

[Nil93] Nilsson, U.: Towards a methodology for the design of abstract machines. *Journal of Logic Programming*, 16(1,2): 163–189, 1993.
[PeD82] Pemberton, S. and Daniels, M.: *Pascal Implementation, The P4 Compiler*. Ellis Horwood, 1982.
[Pet95] Pettersson, M.: *Compiling Natural Semantics*. Ph.D. thesis, Linköping University, Sweden, 1995.
[Plo81] Plotkin, G.: A structural approach to operational semantics. Technical Report DAIMI FN 19, Computer Science Department, Aarhus University, 1981.
[Sch95] Schmidt, D. A.: Natural-semantics-based abstract interpretation (preliminary version). In A. Mycroft, editor, *Proceedings of Second International Symposium on Static Analysis*, volume LNCS 983. Springer-Verlag, 1995.
[Sun95] Sun Microsystems. *Documentation of the Java Developers Kit – version 1.0 Beta*, 1995. Available at http://java.sun.com/JDK-beta/index.html.
[War77] Warren, D. H. D.: Implementing Prolog: Compiling predicate logic programs. D.A.I Research Report No. 40, Edinburgh, 1977.
[WiM92] Wilhelm, R. and Maurer, D.: *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer-Verlag, Berlin, Heidelberg, 1992. English edition: Compiler Design, Addison-Wesley, 1995.
[Win93] Winskel, G.: *The Formal Semantics of Programming Languages*. MIT Press, 1993.

Received May 1997
Accepted in revised form May 2000 by W. L. Scherlis