Semantics-Directed Generation of Compilers and Abstract Machines

Dissertation zur Erlangung des Grades des Doktors der Naturwissenschaften der Technischen Fakultät der Universität des Saarlandes

Stephan Diehl, diehl@cs.uni-sb.de

Computer Science Department, University of the Saarland, Germany
Fachbereich Informatik, Universität des Saarlandes

Saarbrücken, 1996

Das Promotionskolloquium fand am 17. Juli 1996 statt.

Dekan der Technischen Fakultät:	Prof. Dr.-Ing Helmut Bley
Berichterstatter:	Prof. Dr. Reinhard Wilhelm
	Prof. Dr. Harald Ganzinger

Acknowledgements

During the work on this thesis the author was a member of the Graduiertenkolleg "Effizienz und Komplexität von Algorithmen und Rechenanlagen" at the University of the Saarland and supported by a grant of the "Deutsche Forschungsgemeinschaft".
I want to thank Reinhold Heckmann, Stephen McKeever and Reinhard Wilhelm who worked their way through drafts of this thesis and provided technical and stylistic comments. The presentation has especially benefited from the careful eye for detail of Reinhold Heckmann.
Thanks for fruitful discussions to Robert Glück, Christian Fecht, Frank Padberg, Kurt Sieber and all those I undeliberately forgot to mention here.

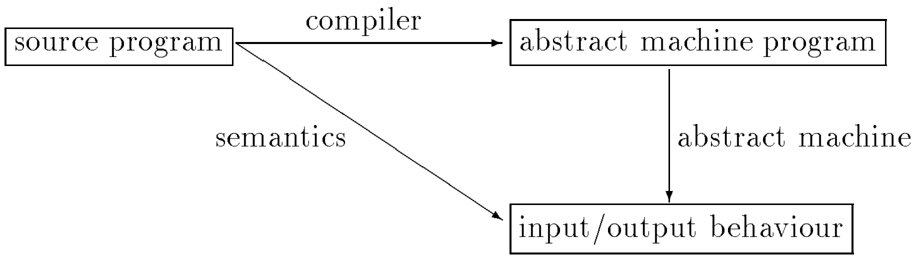
Contents

Chapter 1

Introduction

In traditional compiler design the work of a compiler is divided into several phases: lexical, syntactical and semantical analysis, optimizations and code generation. For several of these phases generators exist – most prominently LEX and YACC for generating lexical and syntactical analyzers. 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, WM92, 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 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.



The behaviour of a program will depend on it's 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. There are various semantic formalisms which are discussed in Chapter 2. In the rest of the thesis we concentrate on natural semantics, but we also address action semantics.

Outline of Thesis

In this thesis we will give an answers to the question, how one can generate translation schemes and abstract machines. It is based on pass separating a natural semantics specification. A second answer was presented in [Dieb]. 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 thesis both guarantees completeness and is fully automatic. As far as we know, our generator is the first running implementation of a system which fully automatically produces both compilers and abstract machines from a semantics specification.

This thesis contains

an overview of semantics formalisms: We illustrate several semantics formalisms and discuss their appropriateness for semantics-directed compiler generation.

an overview of existing work: Work in semantics-directed compiler generation and derivation of abstract machines is based on many formalisms and methods. We elaborate on some fundamental techniques and classify many existing systems and approaches by several criteria. Although such an overview justifies itself, it should also help the reader to classify and appreciate our work.

a generator for abstract machines based on natural semantics: 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 correctness proof of the generator without optimizations: First we present a semantics for our meta-language, then we use it to prove the correctness of several transformations.

an interesting application of the generator: By generating a compiler based on a natural semantics specification of Action Notation we get an action 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 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.

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 to extend the system over time to include more powerful analysis and transformation methods.

Chapter 2

Comparison of Semantic Formalisms

> "The programming language Tower of Babel is well known. Less discussed is the Tower of Metababel, symbolic of the many ways that programming languages are described and defined. The methods used range all the way from natural language to the ultramathematical."
>
> [MLB76]

In this chapter we illustrate several semantic formalisms and discuss their appropriateness for semantics-directed compiler generation. As a running example we use a simple imperative language. Neither do we intend to cover all aspects of the formalisms, nor can we provide an introduction to programming language semantics here. The semantic formalisms are merely described to illustrate some of our critiques.

2.1 The Importance of Doing Formal Semantics

To define a programming language, one has to define what programs in that language look like,i.e., the syntax of the language, and what the meanings of such programs are,i.e., the semantics of the language. To define such meanings, we can use natural language. But often ambiguities arise in natural language descriptions. Thus we need a precisely defined formal notation. Such notations are often called semantic formalisms. The formal description of the semantics of a programming language can for example be used

- as a basis for implementing interpreters or compilers for that language,
- to verify programs,i.e., to prove whether a program meets the specification of its behaviour,
- to prove the equivalence of programs, e.g., of a transformed program and its original form.

If one develops or evaluates a semantic formalism, one has to keep in mind, that there is a variety of concepts in programming languages, which might have to be described by the formalism. Such concepts include variables, control structures, procedures, dynamic storage, data structures, side effects, types, polymorphism, pattern matching, dynamic vs. static binding, call-by-name, call-by-value, call-by-need, higher order functions, logical variables, unification, backtracking, nondeterminism, concurrency, inheritance, objects, messages etc.
In [Mos92] P.D. Mosses describes several requirements for language descriptions from different view points. Language designers need modifiable, extensible and reusable descriptions. For implementors the descriptions should be unambiguous and complete. Programmers, who are going to use the language, prefer easy to understand descriptions, which they can relate to familiar programming concepts. Finally for theoreticians the descriptions should support program verification and have clear and elegant foundations.
Now the question is, what requirements should a language description have from a semantics-directed compiler generation point of view. We will try to answer this question by describing different semantic formalisms and focussing on their appropriateness for automatic generation

of compilers. Although the critiques are partly subjective, other researchers draw similar conclusions [Mos92, Pal92a, Tof90, Lee89, Mou93].

2.2 The Syntax of SIMP

In text books on programming languages often toy languages are presented and used as running examples. Such languages include IMP in [Win93], PROSA in [LMW86] and GRAAL in [Mey90] . For our purposes here we will use a simple imperative language called SIMP which is similar to these.

PROGRAM	::=	$STATEMENT \mid STATEMENT\ \boxed{;}\ PROGRAM$
STATEMENT	::=	$ASSIGNMENT \mid LOOP$
ASSIGNMENT	::=	$VAR\ \boxed{:=}\ EXPRESSION$
EXPRESSION	::=	$VAR \mid INT \mid EXPRESSION\ \boxed{\texttt{-}}\ EXPRESSION$
LOOP	::=	$\boxed{\textbf{while}}\ BOOLEXP\ \boxed{\textbf{do}}\ PROGRAM\ \boxed{\textbf{od}}$
BOOLEXP	::=	$EXPRESSION\ \boxed{>}\ EXPRESSION$

A program in this language is f.e.:

```
X:=10;
while X>0 do X:=X-1 od
```

2.3 Operational Semantics – the SIMP-Machine

Following the approach taken in [LMW86] [1] we define a mathematical machine [2] to give the semantics of SIMP. Basically the SIMP machine consists of configurations and the transition function.

$$M_{SIMP} = (K, K^f, PROGRAM, \delta)$$

A configuration is a pair of a program rest and a binding of variables.

$$K = PR \times B$$

[1] Since SIMP is much simpler than the language described there, we do not need to consider an explicit store and input and output values.

[2] Some authors use the term 'abstract machine' instead, but we use the term 'abstract machines' for an intermediate target language for compilation.

$PROGRAM$ is the set of well-formed programs and $K^f \subseteq K$ is the set of final configurations,i.e., when a compuation reaches a configuration in K^f then it terminates. The transition function δ maps configurations to successor configurations. It is defined by rules like the following ones, where $I(b, E)$ is the interpretation of the expression E in the binding b:

> p is of the form $n := E; p'$ where n is a variable name, E an expression and p' a program rest.
> Then $(p, b) \Rightarrow (p', b[I(b, E)/b[n]])$,
> if $I(b, E)$ is defined.

> p is of the form *while* B *do* p_1 *od*; p' where B is a boolean expression, p_1 a program and p' a program rest.
> Then $(p, b) \Rightarrow \begin{cases} (p_1; p, b) & \mathit{if}\ I(b, B) = \mathit{true} \\ (p', b) & \mathit{if}\ I(b, B) = \mathit{false} \end{cases}$

A computation step is defined as follows:

> Let $k, k' \in K$
> $k \succ k' \Leftrightarrow \delta(k) = k'$

A computation is a possibly infinite sequence $\gamma = (k_1, k_2, ...)$ of configurations where:

$$k_i \succ k_{i+1} \qquad \text{for all } i$$

Finally a terminating computation is a sequence $\gamma = (k_1, k_2, ..., k_n)$ of configurations where:

$$k_1 \succ k_2 \succ \ldots \succ k_n,\ \forall i < n : k_i \notin K^f \text{ and } k_n \in K^f$$

The semantics of a program p is the set of all computations $(k_1, k_2, \ldots)$ where $k_1 = (p, \emptyset)$ and $\emptyset$ is the empty binding.

Pragmatic Considerations: Defining the operational semantics using mathematical machines does not provide for abstractions and modules. In other words the problem is how to combine mathematical machines. Also the meta-language seems to be ad-hoc and would need to be defined precisely. There are also no widely accepted notations and conventions for this way of specifying semantics.

2.4 Structural Operational Semantics and Natural Semantics

Structural Operational Semantics has been proposed by Plotkin [Plo81]. A special style of SOS has become known as Natural Semantics [Kah87].
If we represent configurations as terms, then the semantics of SIMP can be given by inference rules.

$$\frac{\mathrm{(b,E)}\rightarrow\mathrm{V}}{\mathrm{(n{:=}E;p',\ b)}\rightarrow\mathrm{(p',\ b[V/n])}}$$

This rule means: if E in the binding b evaluates to V, then there is a transition from the configuration $(n := E; p',\ b)$ to $(p',\ b[V/n])$.

$$\frac{\mathrm{(b,B)}\rightarrow\mathrm{true}}{\mathrm{(while\ B\ do\ p_1\ od;p',\ b)}\rightarrow\mathrm{(p_1;while\ B\ do\ p_1\ od;p',\ b)}}$$

$$\frac{\mathrm{(b,B)}\rightarrow\mathrm{false}}{\mathrm{(while\ B\ do\ p_1\ od;p',\ b)}\rightarrow\mathrm{(p',b)}}$$

The semantics of a program p is the set of all pairs (b, b') where $(p, b) \rightarrow (\epsilon, b')$ is provable. Note, that if a program p does not terminate for a given binding b, one cannot prove $(p, b) \rightarrow (\epsilon, b')$ for any b'.
The above rules define a small-step semantics of SIMP,i.e., they define transitions from configurations to successive configurations. We can also define a big-step semantics, which defines transitions from configurations to the environment, they will eventually evaluate to, e.g., :

$$\frac{\mathrm{(b,B)}\rightarrow\mathrm{true}\qquad\mathrm{(p_1;while\ B\ do\ p_1\ od;p',b)}\rightarrow\mathrm{b'}}{\mathrm{(while\ B\ do\ p_1\ od;p',b)}\rightarrow\mathrm{b'}}$$

$$\frac{\mathrm{(b,B)}\rightarrow\mathrm{false}\qquad\mathrm{(p',b)}\rightarrow\mathrm{b'}}{\mathrm{(while\ B\ do\ p_1\ od;p',b)}\rightarrow\mathrm{b'}}$$

Big-step semantics is often called Natural Semantics and we will use this term throughout this thesis. We will discuss Natural Semantics later in more detail (see Chapter 4).
Pragmatic Considerations: SOS is in general not compositional. Compositionality would allow for generating different parts of a compiler independently. Moreover SOS does not

enforce determinism or confluence. This might be helpful to specify nondeterministic or concurrent languages, but can be a hinderance when defining deterministic languages. SOS specifications are not very maintainable, e.g., if we want to add a new component to the program state, we have to change the state in all rules. Despite of all pragmatic defects of SOS semantics, they have been used quite a lot, e.g., to specify type systems or semantics of programming languages, most notably ML [MTH90].

2.5 Evolving Algebras

Since evolving algebras are a relatively new formalism not covered in most books on programming language semantics, we will give a more detailed description here.
Evolving algebras (EvAs) have been proposed by Gurevich in [Gur91] and used by Gurevich and others to give the operational semantics of languages like C, Modula-2, Prolog and Occam. Börger and Rosenzweig's proof of the correctness of the Warren Abstract Machine is based on a slight variation of evolving algebras ([BR92]). An evolving algebra may be tailored to the abstraction level necessary for the intended application of the semantics, e.g., we might have a hierarchy of evolving algebras, each being more concrete with respect to certain aspects of the semantics.
For our purposes here, we need a precise definition of what an EvA is, and what a computation of an EvA looks like. Given a set M we define $M^* = M \cup (M \times M) \cup (M \times M \times M) \cup \ldots$

Definition: An evolving algebra Ψ is a quadruple $< \sigma, S, T, \mathcal{I}_0 >$ where

- σ is a **signature**,i.e., a finite set of function names with associated arity
- S is a nonempty set, called the **superuniverse**[3]
- T is a finite set of transition rules
- $\mathcal{I}_0 : \sigma \to (S^* \to S)$ is the initial interpretation of functions in σ,i.e., $\mathcal{I}_0$ maps every function name f of arity n to an interpretation function $\mathcal{I}_0(f) : S^n \to S$.

Transition rules[4] are of the form:

function update: $\boxed{f(t_1, ..., t_n) := t_0}$
where $f \in \sigma$, $n \geq 0$ is the arity of f and the t_i are terms

[3] We will assume $\{true, false\} \subseteq S$
[4] We will only consider finite terms and finite transition rules.

guarded update: $\boxed{if\ b\ then\ C}$
where b is a term and C is a set of transition rules

A term t is an element of S or has the form $f(t_1, ..., t_n)$, where $f \in \sigma$,$n \geq 0$ is the arity of f and the t_i are terms.
We will use the notation $\mathcal{I} \xrightarrow{\Psi} \mathcal{I}'$ to indicate, that $\mathcal{I}'$ is the result of simultaneously applying the transition rules of Ψ to $\mathcal{I}$. We will call this a **step** of the evolving algebra. Before we can define a step of an EvA, we have to introduce some notation:

$$\begin{aligned}
eval(f(t_1, ..., t_n), \mathcal{I}) = \quad & \mathcal{I}(f)(eval(t_1, \mathcal{I}), ..., eval(t_n, \mathcal{I})) \quad for\ n \geq 0 \\
eval(f(t_1, ..., t_n) := t_0, \mathcal{I}) = \quad & f(eval(t_1, \mathcal{I}), ..., eval(t_n, \mathcal{I})) := eval(t_0, \mathcal{I}) \quad for\ n \geq 0 \\
\\
updates(T, \mathcal{I}) = \quad & \{eval(u, \mathcal{I}) | u \in T \wedge u \text{ is a function update}\} \\
& \cup\ updates(\bigcup_{\substack{\boxed{if\ b\ then\ C} \in T \\ \wedge\ eval(b, \mathcal{I}) = true}} C, \mathcal{I})
\end{aligned} \tag{2.1}$$

Let M be a set of evaluated function updates, then $\overline{M}$ denotes the set of all greatest subsets A of M, such that if $\boxed{f(t_1, ..., t_n) := t_0}$ in A then there is no update $\boxed{f(t_1, ..., t_n) := t_0'}$ in A where $t_0 \neq t_0'$. The relation $\xrightarrow{\Psi}$ is defined as follows:

$$\mathcal{I} \xrightarrow{\Psi} \mathcal{I}' \Leftrightarrow \begin{array}{l} \exists U \in \overline{updates(T, \mathcal{I})}\ \forall \tilde{a} \in S^*, f \in \sigma : \\ \mathcal{I}'(f)(\tilde{a}) = \begin{cases} t & if\ \boxed{f(\tilde{a}) := t} \in U \\ \mathcal{I}(f)(\tilde{a}) & otherwise \end{cases} \end{array}$$

Note, that if $\overline{updates(T, \mathcal{I})}$ is not a singleton, then from every set of conflicting updates only one member is chosen nondeterministically.

A **computation** of an evolving algebra Ψ is a sequence $\gamma = <\mathcal{I}_0, \mathcal{I}_1, ..., \mathcal{I}_k>$, such that $\mathcal{I}_0 \xrightarrow{\Psi} \mathcal{I}_1 \xrightarrow{\Psi} ... \xrightarrow{\Psi} \mathcal{I}_k$. If $updates(T, \mathcal{I}_k) = \emptyset$ then γ is a terminating computation.
Here are some transition rules of an EvA for SIMP. In contrast to other semantic formalisms, there is no pattern matching in the transition rules of an evolving algebra. Instead we have to define accessor functions like hd which returns the first element of a list and tests like $is\_assignment$, which is true, if its argument is a representation of an assignment. If its argument is an assignment then $left\_val$ yields its left hand side, and $right\_val$ its right hand side.

```
if is_assignment(hd(pr)) then
 { b(left_val(hd(pr))) := eval(b,right_val(hd(pr))),
   pr := tl(pr)
 }

if is_loop(hd(pr)) then
  { if eval(b,condition(hd(pr)))=true then
     { pr := cons(body(hd(pr)),pr)
     }
    if eval(b,condition(hd(pr)))=false then
     { pr := tl(pr)
     }
  }
```

In this EvA `pr` and `b` are dynamic functions,i.e., are modified by function updates, whereas `hd`, `tl`,`cons`,`is_loop`, ... are static functions. The semantics of a program p is the set of all computations where p is the initial value of `pr` and the value of `b` is a binding.
A pass separation transformation is presented and proved correct in [Die95a] and can be seen as a first attempt towards SDCG based on evolving algebras.
Pragmatic Considerations: All rules can be applied at the same time, so one has to make sure by nested guarded updates, that the right update is executed. As a result the rules get long and complicated. Furthermore Evolving Algebras are not yet widely used. On the other hand specifications are usually very operational and thus might lead to efficient implementations.

2.6 Denotational Semantics

Denotational Semantics maps each program construct to a mathematical object, its denotation. The mapping is defined by semantic equations. In our example the denotations are functions form bindings to bindings.

$$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

$$\mathcal{C}[\![X := a]\!] = \lambda b.b[v/X]) \qquad \textit{where } v = \mathcal{E}[\![a]\!]b$$

Now the denotation of the WHILE loop has to be a function, too. But the existence of such a function is not obvious at all. Analoguous to the definitions in the previous sections we define the semantics of the WHILE loop by copying a part of the program text:

$$\mathcal{C}[\![while\ e\ do\ c\ od]\!] = \lambda b.if\ \mathcal{E}[\![e]\!]b = true\ then\ \mathcal{C}[\![c; while\ e\ do\ c\ od]\!]b\ else\ b$$

But is this well-defined ? Unfolding $\mathcal{C}[\![c; while\ e\ do\ c\ od]\!]$ yields $\mathcal{C}[\![while\ e\ do\ c\ od]\!] \circ \mathcal{C}[\![c]\!]$ and thus the above definition is recursive. If we read the definition as an equation, then the question arises whether it has a solution: What values ψ of $\mathcal{C}[\![while\ e\ do\ c\ od]\!]$ satisfy the equation. Here denotational semantics resorts to fixed-point theory. It can be shown that the smallest fixed-point of the following function is the most general,i.e., least informative solution, of the above equation.

$$\Gamma(\psi) = \lambda b.if\ \mathcal{E}[\![e]\!]b = true\ then\ (\psi \circ \mathcal{C}[\![c]\!])b\ else\ b$$

and thus

$$\mathcal{C}[\![while\ e\ do\ c\ od]\!] = fix(\Gamma)$$

The semantics of a program p is simply $\mathcal{C}[\![p]\!]$,i.e., a function mapping bindings to bindings.
Pragmatic Considerations: The basic operations of λ-notation (e.g., abstraction, application, tupling) do not correspond to fundamental concepts of programming languages These basic operations are suitable for functional languages, but they lead to much overhead when specifying imperative or logical languages. This point is discussed at length by Mosses [Mos92] and has been a motivation for the development of Action Semantics. Another problem with denotational semantics is that there are too few basic operations, such that specific transformations and optimizations are difficult. In [Lee89] Peter Lee gives a more detailed critique of denotational semantics, e.g., he discusses its lack of separability,i.e., the intertwining of model-dependent details and the actual semantics. As an example he mentions that many denotational semantic specifications dictate the structure of environments, but in the semantics we only need their functionality. If we could describe their functionality independently, then we could replace environments by different structures. This leads to another weakness of denotational semantics, namely its lack of modularity.

2.7 Axiomatic Semantics

Axiomatic semantics use partial correctness assertions of the form $\{A\}\ c\ \{B\}$. The interpretation of such an assertion is: "If property A holds before the execution of program c and the execution of c terminates, then property B holds after its execution." The semantics of SIMP is described by the following inference rules:

$$\frac{\{\mathrm{A}\}\ \mathrm{c}_0\ \{\mathrm{B}\} \quad \{\mathrm{B}\}\ \mathrm{c}_1\ \{\mathrm{C}\}}{\{\mathrm{A}\}\ \mathrm{c}_0;\mathrm{c}_1\ \{\mathrm{C}\}}$$

$$\{B[a/X]\}\ X := a\ \{B\}$$

$$\frac{\{\mathrm{A}\wedge\mathrm{b}\}\ \mathrm{c}\ \{\mathrm{A}\}}{\{\mathrm{A}\}\ \mathrm{while\ b\ do\ c\ od}\ \{\mathrm{A}\wedge\neg\mathrm{b}\}}$$

An important, language independent rule, also known as rule of consequence, is:

$$\frac{(\mathrm{A}\Rightarrow\mathrm{A}') \quad \{\mathrm{A}'\}\ \mathrm{c}\ \{\mathrm{B}'\} \quad (\mathrm{B}'\Rightarrow\mathrm{B})}{\{\mathrm{A}\}\ \mathrm{c}\ \{\mathrm{B}\}}$$

It states that less informative properties can be deduced from ones that carry more information. The semantics of a program p is the set of all pairs (P, Q), where $\{P\}S\{Q\}$ is provable by means of the axioms.
Pragmatic Considerations: Usually axiomatic semantics is designed to prove properties of programs, but does not necessarily provide enough information to implement a language based on the specification. The actual semantics of the programming language is only one model for the given axiomatic semantics. There are many other models,e.g., the semantics which diverges (does not terminate) for every language construct. This is due to the fact that axiomatic semantics focusses on partial correctness, only. It is also not clear, what language should be used to write properties. Often concepts from the programming language are lifted into the properties,e.g., program variables are used to represent the values they are bound to: $\{x < 0\}\ x := x * x\ \{x > 0\}$

2.8 Action Semantics

Research in the field of denotational semantics has lead to the development of action semantics [Mos92]. Originally combinators, i.e., macros for λ-expressions, have been used to make denotational semantics descriptions more readable. In High-Level Semantics [Lee89] operators have been added to the formalism. The semantics of those operators are given by an interpretation,i.e., a mapping of the operators to functions.
In [Mos83, Mos84] Peter Mosses introduces abstract semantic algebras to allow for operations in semantic descriptions which correspond to fundamental concepts of programming languages. In action semantics, these operators are called actions and action combinators, and their semantics is given by SOS rules[5]. Similar to denotational semantics, the action semantics of a programming language is given by semantic equations:

[5]Actually Peter Mosses defined these actions algebraically in [Mos92].

(1) execute⟦ X ":=" E ⟧ = | evaluate E
then store the value in the cell bound to X

(2) execute⟦ "while" E "do" C "od" ⟧ = unfolding
| | evaluate E
| then
| | | execute C then unfold
| | else
| | | complete

Although the right hand sides of these equations look like English phrases, they are written in a formal language called action notation.
The formal definitions of some of the actions and action combinators in the above action semantic description are given below. Here t, t', t_i are bindings for transient data, b, b', b_i are bindings of scoped data and s, s', s_i are stores. Execution of an action can complete or fail, this is indicated by the outcome status o, o'.

$$\frac{(\mathrm{t,b,s} \rightarrow \mathrm{a[unfolding\ \ a/unfold]}) \Rightarrow (\mathrm{o',t',b',s'})}{(\mathrm{t,b,s} \rightarrow \mathrm{unfolding\ \ a}) \Rightarrow (\mathrm{o',t',b',s'})}$$

unfolding a proceeds by replacing every occurence of *unfold* in the action a by *unfolding a* and then executing the resulting action notation term.

$$\frac{(\{\},\mathrm{b,s} \rightarrow \mathrm{a_1}) \Rightarrow (\mathrm{o_1,t_1,b_1,s_1})}{(\{0 \Rightarrow \mathrm{true}\}.\mathrm{t,b,s} \rightarrow \mathrm{a_1\ \ else\ \ a_2}) \Rightarrow (\mathrm{o_1,t_1,b_1,s_1})}$$

$$\frac{(\{\},\mathrm{b,s} \rightarrow \mathrm{a_2}) \Rightarrow (\mathrm{o_2,t_2,b_2,s_2})}{(\{0 \Rightarrow \mathrm{false}\}.\mathrm{t,b,s} \rightarrow \mathrm{a_1\ \ else\ \ a_2}) \Rightarrow (\mathrm{o_2,t_2,b_2,s_2})}$$

If the first transient given bz the action a_1 is *true*, then the result of performing a_1 is the result of performing a_1 *else* a_2. Otherwise, if the first transient is *false* then the result is that of performing a_2.

Using the semantics equations a program p is transformed into an action notation term t. One can regard this action notation term as the semantics of the program p. Alternatively, one could also define the semantics of the term t with respect to the above SOS specification as the semantics of p. We will give a more detailed introduction to Action Semantics in Section 7.1 and a SOS specification of Action Semantics in Appendix B.

Pragmatic Considerations: Action notation is a rich, abstract, modular, and compositional language. Its operations are close to the concepts of existing programming languages (mostly functional, imperative, and concurrent languages). A major drawback is that it is not that easy to learn, since for each action one has to know exactly what its effects are, e.g., how it modifies transients, bindings and the store. Although action semantics specifications are very readable, one has to be aware of what happens below the surface.
Since Action Semantics is still in its infancy, it is not yet widely used and it is only fair to say that up to now one cannot seriously predict its success.

2.9 Translational Semantics

The semantics of a programming language can be defined by translating its constructs to constructs in another programming language with a given semantics. A special case of this approach is to define new constructs by constructs of the same language, which have already been defined (bootstrap method). We will use $\langle\ldots\rangle$ as a constructor for instruction lists and the dot as an infix operator for appending such lists.

$$
\begin{array}{l}
trans[\![X'' := '' E]\!] \;= trans[\![E]\!].\langle STORE\ X\rangle \\
trans[\![''while''\ E\ ''do''\ C\ ''od'']\!] \;= \\
\quad let \\
\qquad l1 = newlabel(),\ l2 = newlabel(), \\
\qquad e = trans[\![E]\!], c = trans[\![C]\!] \\
\quad in \\
\quad \langle l1\rangle.e.\langle ELSE\ l2\rangle.c.\langle GOTO\ l1,\ l2\rangle
\end{array}
$$

Depending on what we need it for, the semantics of a program p is the target language program $trans[\![p]\!]$ it is translated to or the semantics of $trans[\![p]\!]$.
Pragmatic Considerations: Obviously this is the best starting point for compiler generation, because the compiler is explicitly given. But such specifications are hard to read and write. Nevertheless a translational semantics can be of advantage as one stage in a semantics-directed compiler generator. We could translate a full-fledged language into a core language for which the semantics is given using a different approach.

2.10 Conclusions

Different semantic formalisms are suitable for different purposes. Some programming language concepts can be defined more easily in one formalism than in an other, e.g., nondeterminism is easy to define using SOS rules, but much harder if one uses the denotational method. For the purpose of semantics-directed compiler generation such formalisms are preferable, which are close to real programming languages. Furthermore compositionality could enable the generator to produce different parts of the compiler independently. In summary, drawbacks of different formalisms have been that they are vague, adhoc, not compositional, lack readability, separability or modularity and that their underlying concepts are too far from real computers.

Chapter 3

Existing Work in Semantics-Directed Compiler Generation

Work in semantics-directed compiler generation and derivation of abstract machines is based on many formalisms and methods. We elaborate on some fundamental techniques and classify many existing systems and approaches by several criteria.

3.1 Introduction

There is a huge amount of existing work in semantics-directed compiler generation (SDCG). Our goal is to evaluate relevant work in the field and to show some of the fundamental concepts. We can only consider a small part of all the existing work, but we chose more recent, different and promising approaches [1]. We also address some approaches to derive abstract machines, thus providing the background for the new approach presented in this thesis. Finally we point out some open research problems related to SDCG.

3.2 What is SDCG ?

Let us start by giving an informal definition of what a semantics-directed compiler generator is:

[1] Pioneer work can be found in [Jon80], an early survey is given in [Gau81]. Some systems are also surveyed in [Sch86] and [Tof90].

> A semantics-directed compiler generator is an algorithm, which given a definition of the semantics of a source language produces a compiler for that language.

Of course, the generated compiler should preserve the semantics of the source language in a reasonable sense, e.g., as in the definition of [CJ83].
SDCG is part of the more general problem of generating programming language tools from formal specifications [2]:

specification	generated tool or compiler phase
dynamic semantics	interpreter, debugger, code generator
static semantics	static analyzers (e.g., type checker)
source-to-source transformations	optimizer
syntax	parser, pretty printer, structure editor

3.3 Advantages of SDCG

Although there has been a lot of research in that field, there are no widely used semantics-directed compiler generators. The major motivation for the interest in semantics-directed compiler generators is their advantage over handwriting compilers.

Correctness: Assuming the generator has been verified, the generated compilers are automatically or implicitly proved correct.

Readability: It is easier to write the specification of a programming language than writing a compiler. These specifications are also more readable and intelligible.

Maintainability: It is much easier to extend or change an existing specification and generate a new compiler than extending or changing an existing compiler.

Portability: By porting the compiler generator or changing the definition of the target language, we can generate compilers for different architectures. We don't have to change the specifications of the source languages.

Insight: SDCG is often based on semantics preserving transformations. These transformations relate source language constructs to target code. Tracing these transformations, we can explain why certain target code is produced by the generated compiler.

[2] In general, for a practical tool we need several such phases, e.g., an interpreter might include a parser and a static analyzer.

One might even argue that it is a question of personal taste, whether to use handwritten compilers or generated ones. But there are critical applications, where correctness is a matter of urgent necessity (e.g., nuclear power plants, automated public transportation, automated lasers for medical treatment) and proving handwritten compilers correct is extremely difficult (e.g., [Rus92]). In contrast, once the compiler generator has been proved correct, all generated compilers are correct with respect to their source language specification. Unfortunately, such proofs are rare (e.g., [Pal92a],[JGS93]).

3.4 Disadvantage of SDCG

Unfortunately, in general we cannot expect a generated compiler to be as efficient as a handwritten one. A compiler writer uses optimizations, which are based on experience, insight and common sense [3]. Performance results as those of the Action Semantics-directed compiler generator OASIS [Orb94], although encouraging, have been merely possible because of the small semantic distance of the source and specification language and the use of a handwritten, optimizing compiler for the specification language. In particular, many source language constructs have direct counterparts in the specification language. The semantic distance of the specification and the target language is much greater, but it is bridged by the handwritten compiler.

3.5 Criteria

To evaluate the existing work, we chose criteria such as **source language** or **transformations**, which we found to be important for describing what a certain system does. First we have to be more precise, what we mean by these criteria:

Automatable (aut) Is there a running implementation or has the method including all heuristics been formalized to such an extent that it can be implemented?

Correctness (corr) Is there a correctness proof of the method or the system?

Efficiency (eff) If there is a system, then does it produce the compiler fast, does the compiler compile fast and is the compiled program fast? In Tables 3.1 and 3.2 an entry such as -,?,3 means, that the generation is inefficient (-), we have no information on the efficiency

[3] Common Sense Knowledge and Reasoning has turned out to be the hardest problem in symbolic AI (artificial intelligence).

Author	**Quality**	**Languages**	**Transformations**	**Papers**
Peter Lee, U.F. Pleban	aut: MESS corr: no eff: +,+,0	spec: High-Level Semantics obj: Scheme src: C, Pascal trg: prefix-form operator expression	Basically: unfolding and partial evaluation of the macro-semantics	[Lee89], [PL88], [PL87]
N.D. Jones, Mads Tofte	aut: CERES corr: no eff: ?,?,2	spec: control flow language with operators and interpretation of these obj: similar to specification language, but only GOTOs for control flow src: a toy language, Prolog trg: same as object language	composition and compilation	[Tof90], [CJ83]
Martin R. Raskovsky	aut: ISL corr: no eff: ?,?,0	spec: extended typed λ calculus (denotational semantics) obj: subset of BCPL src: C trg: DEC-10 code	$>$ 40 transformation rules	[Ras82]
Richard Kelsey, Paul Hudak	aut: yes corr: no eff: ?,+,0	spec: call-by-value λ calculus with implicit store obj: Scheme src: Pascal, Basic trg: a sub-language of the source language which has neither calls nor nested scoping	linearization, adding continuations and environment, register allocation	[Kel89], [KH89]
F. Nielson , H. R. Nielson	aut: PSI corr: yes eff: ?,?,3	spec: denotational semantics written in TML_S obj: abstract interpreter src: none trg: code for a functional abstract machine	compilation of TML_S by abstract interpretation	[NN86]
Mads Dam, Frank Jensen	aut: yes corr: yes eff: ?,?,-	spec: structural operational semantics and interpretation of operations obj: translation schemes src: a toy language trg: imperative language	generate stack semantics, make deterministic, detect recursion, produce schemes	[DJ86]

Comments:

CERES: First the language specification is composed with a specification of the translation from the specification language to semantic expressions. Next the composed definition is translated to a semantic expression and then into the object language

MESS: High-Level Semantics consists of macro-semantics (= denotational semantics with operators) and micro-semantics (= definitions of operators)

Table 3.1: Approaches to SDCG (part I)

Author	Quality	Languages	Transformations	Papers
Jens Palsberg	aut: CANTOR corr: yes eff: ?,?,2	spec: action semantics specification obj: Scheme src: a toy language trg: code for an abstract RISC machine	type analysis, compilation (basically: unfolding of actions)	[Pal92a]
Peter Orbæk	aut: OASIS corr: no eff: +,+,0	spec: action semantics specification obj: Perl, Scheme, C++ src: functional, imperative and OO languages trg: SPARC code	type and other analyses, compilation (basically: unfolding of actions) and optimizations	[Orb94]
D.F. Brown, H. Moura, D. A. Watt	aut: ACTRESS corr: no eff: ?,?,2	spec: action semantics specification obj: ML src: functional and imperative languages trg: C	sort checking, transformations on actions, C code generation	[BMW92], [MW94]
N.D. Jones, C.K. Gomard, P. Sestoft	aut: self-applicable partial evaluators corr: yes (for some partial evaluators) eff: ?,?,-	spec: interpreter in Flow Chart language, C, Prolog or Scheme obj: compiler or compiler generator in specification language src: Turing Machine, assembly language, Prolog trg: same as specification language	partial evaluation of an interpreter or partial evaluator	[JGS93], [Ses86]
Charles Consel , Siau Cheng Khoo	aut: in part: SCHISM = partial evaluator corr: no eff: ?,?,-	spec: Scheme function definitions based on denotational semantics obj: Scheme src: Prolog trg: Scheme (residual program = compiler for a query)	partial evaluation of the Prolog program (static) and the query (dynamic)	[CK91]

Comments:

CANTOR: In [BP93] the authors replace the handwritten action compiler of the CANTOR system by an action compiler which was generated by partially evaluating an action interpreter written in SCHEME using the SIMILIX partial evaluator.

Table 3.2: Approaches to SDCG (part II)

of the compiler (?), and the compiled program is 3 orders of magnitude slower than a program compiled by a handwritten compiler.

Specification Language (spec) What language or form is used to define the semantics of the source language?

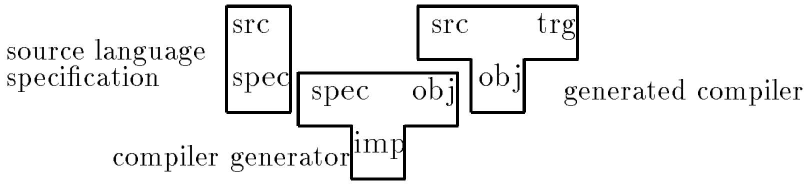
Object Language (obj) What language or form is used to represent the generated programs, esp. compilers?

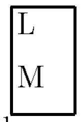
Source Language (src) To which languages has the system or method been applied, esp. for which languages have compilers been generated?

Target Language (trg) What language does the generated compiler produce code in ?

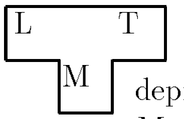
Transformations What transformations are used to generate a program (esp. compiler) from the specifications of the source language?

An overview of existing systems and approaches to SDCG is given in Table 3.1 and 3.2. The relation of the different languages involved is shown below using T-diagrams[4]:





depicts an interpreter written in the language M for the language L.



depicts a compiler written in the language M. The compiler converts programs written in the language L into equivalent programs in the language T.

[4]Here **imp** is the implementation language of the compiler generator.

3.6 Specification Languages

As noted by Tofte [Tof90, page 33], if a specification of a language suffices to generate a specification of a compiler for that language, then it must itself be a specification of a compiler. The only difference are the languages these specifications are written in. Thus if the specification is f.e. written in action semantics notation, we just give the specification a new reading. Finding this new reading is what SDCG is all about. Depending on the specification language, this might be easy or not. Next we will describe, how certain specification languages have been used as a basis for SDCG.

Standard Programming Languages An interpreter is written in a programming language and then transformed [JGS93, JS86, Kur87, Nil93, Die93]. In most of these approaches, partial evaluation plays a central role.

Denotational Semantics Source programs are converted into λ-expressions according to the denotational specification of the source language. These λ-expressions are reduced as far as possible. At runtime these reduced expressions are further reduced with respect to the runtime input. In addition to these reductions, some approaches use techniques like conversion to continuation-passing style and translation of the λ-expressions to some machine code [Ras82, KH89, CK91, Set82, Wan84, Sch86].

Two Level Semantics The denotational semantics is extended by operators and an interpretation for those operators. Expressions are only reduced up to these operators. The operators are implemented according to the given interpretation as in Peter Lee's High-Level Semantics [Lee89, Wei87, Tof90, NN86]. Using efficient implementations of those operators and even a code generator for expressions composed of operators Peter Lee developed a realistic compiler generator.

Action Semantics Action Semantics is close to two level semantics. The operators are called actions here. The important point about action semantics is the clever choice of actions provided and the algebraic laws, which apply to these actions [Pal92a, Orb94, BMW92, MW94]. Furthermore partial evaluation has been used to compile actions [BP93].

Structural Operational Semantics, Natural Semantics Most of the work based on this kind of semantics is not implemented [DJ86, Han91a, HM90], instead semantics specifications are transformed by hand. In rare cases the correctness of the transformations is proved as in John Hannan's work [5] TYPOL (c.f. [Des84]) is an implementation of a semantic prototyping system based on natural semantics. In [Has88] the author describes partial evaluation of TYPOL specifications and mentions, that it can be used to generate compilers. Recently Pettersson developed an efficient compiler for natural semantics specifications [Pet94, Pet95]. The compiler uses continuation-passing style as an intermediate representation and produces portable C code.

Attribute Grammars As proposed in [Knu68], attribute grammars can be used to define the semantics of programming languages and there are several compiler-generation systems based on attribute grammars, e.g., MUG2 [GRW77], GAG [KHZ82], HLP [Räi80], PCG [Pau82]. For an overview see [DJL88].

Other Systems Other semantic-based systems use algebraic specifications, e.g., Perluette [Gau83], or as in [SV84] a special meta-language CAT is developed, which is almost the union of all the respective source languages.

Some of the specification languages are typed. Type analysis helps to find errors in the specification. Moreover knowing the type of a variable can remove type checks at runtime, thus resulting in more efficient target programs. Note that this is not the same as having a type system for the source language.

3.7 Partial Evaluation

Assume we divide the input of a program p into two components (s, d): a static and a dynamic one. The static data are known at partial evaluation time of the program, whereas the dynamic data are not known before running the residual program $r = peval(p, s)$, i.e., the result of partially evaluating the original program. Partial evaluation replaces all expressions in the program, which only depend on the static data, by the result of their evaluation. Finding those expressions, which only depend on static data is called binding-time analysis. In the following simple example we assume that `y` is static and its value is 3:

```
if y>2 then add(x,sub(y,1)) else y
```

[5] Except for his pass separation transformation all other transformations are specific for the natural semantics specification of a call-by-value λ-calculus.

Since we know the value of `y`, we can decide which alternative of the conditional has to be evaluated. We can also evaluate `sub(y,1)`, but `add(x,sub(y,1))` also depends on the dynamic data `x`. Thus the partial evaluation of the above expression yields:

```
add(x,2)
```

In many compiler generators unfolding and computation of functions with all arguments known play a central role. In the following example we use a functional meta-language to specify the semantics of SIMP. Here, ρ is an environment which maps variable names to addresses in the store σ.

$$
\begin{array}{ll}
eval(i, \rho, \sigma) = i & (i \text{ is an integer}) \\
eval(x, \rho, \sigma) = \sigma(\rho(x)) & (x \text{ is a variable}) \\
eval(e_1 + e_2, \rho, \sigma) = eval(e_1, \rho, \sigma) + eval(e_2, \rho, \sigma) &
\end{array}
$$

$$
\begin{array}{l}
exec(s; p, \rho, \sigma) = eval(p, \rho, eval(s, \rho, \sigma)) \\
exec(v := e, \rho, \sigma) = \sigma[eval(e, \rho, \sigma)/\rho(x)]
\end{array}
$$

$$
\begin{array}{l}
exec(while\ b\ do\ c\ od, \rho, \sigma) = \\
\quad if\ eval(b, \rho, \sigma)\ then \\
\quad\quad exec(while\ b\ do\ c\ od, \rho, exec(c, \rho, \sigma)) \\
\quad else\ \sigma\ fi
\end{array}
$$

Let us write $\$n$ for addresses in the store. Unfolding the call $exec(while\ x < 10\ do\ x := x + 1\ od, \rho, \sigma)$ in a static environment $\rho = \{x \mapsto \$1\}$ but unknown store σ we get

$$
\begin{array}{rl}
resid(\sigma) = & if\ \sigma(\$1) < 10\ then \\
& \quad resid(\sigma[\sigma(\$1) + 1/\$1]) \\
& else\ \sigma\ fi
\end{array}
$$

What is important about this residual program is that the *while*-loop has been translated into the same language the interpreter was written in.

Partial evaluation of different programming languages is covered in [JGS93]. The above example shows two important techniques used in partial evaluators: **Unfolding** of function calls, i.e., replacing a function call by the according instance of the function definition, and **program point specialization**, i.e., defining specialized versions of a function and replacing calls to the function by calls to the according specialized function.

Partial evaluation can be used in several ways in SDCG. First the generated compiler c can just be a partial evaluator, where the program is a fixed interpreter i, the static data are the input programs for that interpreter and the dynamic data are the inputs for those programs: $c = peval(i, .)$ i.e., $c(p) = peval(i, p)$. This way of generating a compiler is an instance of the 1st Futamura Projection [Fut71]. Second a compiler can be generated by partially evaluating a partial evaluator with respect to an interpreter: $c = peval(peval, i)$, which is often referred to as the 2nd Futamura Projection [6].

3.8 Special Transformations

We show some special transformations by means of examples. These transformations have been used by different authors for different specification languages. Although being very simple, the examples should convey the basic ideas.

3.8.1 Linearization

Linearization is a transformation which makes the evaluation order of arguments explicit. The following is an example of an arithmetic expression.

$$add(sub(x, y), mult(v, w))$$

We use λ-abstraction to enforce left to right evaluation of the arguments in a call-by-value λ-calculus:

$$(\lambda a.((\lambda b.add(a, b))mult(v, w)))sub(x, y)$$

Note that by beta-reduction we get the previous form.

3.8.2 Conversion to Continuation-Passing Style

For each basic function f we construct a new function f' which additionally accepts as its first argument a continuation, e.g., : $sub'\ c\ (a, b) = c(sub(a, b))$ or short $sub' = \lambda c.c \circ sub$
By id we denote the neutral continuation. Conversion of the above expression to continuation-passing style yields:

$$\begin{array}{lll} sub' & (\lambda a.mult' & (\lambda b.add'\ id(a, b)) \\ & & (v, w)) \\ & (x, y) & \end{array}$$

[6] Finally the 3rd Futamura Projection tells us how to generate a compiler generator: $cogen = peval(peval, peval)$.

The advantage of continuation-passing style (CPS) is that control flow and data flow are represented in a well-defined, uniform way (c.f. [App89, App92]). In [Kel95] Richard Kelsey shows that CPS can be converted into static single assignment form (SSA) and vice versa. SSA is a program representation, which has been widely used for data-flow analysis and optimizations of imperative programming languages.
After some control flow analysis we might get target code similar to the following, admittedly unefficient code [7]:

```
main: load   x,R1    mcont: load   v,R1    acont: load   a,R1
      sub    y,R1           mult   w,R1           add    b,R1
      store  R1,a           store  R1,b           return
     (goto   mcont)        (goto   acont)
```

3.8.3 Making a Program Deterministic

We now turn our attention to a technique which has been used to transform natural semantics or structural operational semantics specifications (e.g., in [HM90]). These specifications allow for nondeterminism. Nondeterminism comes into play, when a conclusion depends on several preconditions and the order to prove these preconditions is not specified. To make those specification deterministic, search is made explicit by introducing a proof stack. We will again demonstrate this technique by a simple example [8] :

$$\frac{\text{eval(X,A)} \qquad \text{eval(Y,B)}}{\text{eval(plus(X,Y),C)}} \quad \text{C} = \text{A} + \text{B}$$

This rule states, that $plus(X, Y)$ evaluates to the sum of A and B, if X evaluates to A and Y evaluates to B. But there is no order on the preconditions. The above rule is now transformed into

$$\frac{\text{prove(eval(X,A)::eval(Y,B)::R)}}{\text{prove(eval(plus(X,Y),C)::R)}} \quad \text{C} = \text{A} + \text{B}$$

According to the new rule, to prove that $plus(X, Y)$ evaluates to the sum of A and B, we first have to prove that X evaluates to A and then that Y evaluates to B [9].

[7] For complete and more sophisticated examples see [KH89, Kel89]. In Kelsey's PhD thesis further transformations like adding environments greatly improve the quality of the generated code. Actually by the clever choice of the intermediate language all the transformations in his thesis are source-to-source transformations.

[8] This transformation does not deal with another kind of nondeterminism, which arises when several rules are applicable at a time. We attack this problem in Section 5.3.3.

[9] Actually all rules have to be modified in a way such that their conclusions only refer to the first goal on the proof stack.

3.9 Realistic Compiler Generation

We call a compiler generator **realistic**, if the code produced by the generated compilers executes almost as fast as the code produced by handwritten compilers. Among these systems are SAM, MESS, and OASIS. Common features of these are:

- they offer a rich set of primitive operations, or the user can define new operations and provide an efficient implementation of these. In particular operations for environment lookup, memory management and handling of recursion are provided.
- the generated compilers use the semantic equations to transform a source language program into an intermediate form (e.g., action term in OASIS, prefix-operator expression in MESS). Then a handwritten code-generator produces efficient target code for this intermediate representation.

3.10 A more complete Example

In the following we will turn to Action Semantics. Our example is based on [BMW92]. We won't explain Action Semantics here. A natural semantics specification of Action Semantics is given in Appendix B. First we give a fragment of a semantic specification dealing with arithmetic expressions.

```
(1)  evaluate[[I:Integer]] = give IntegerValuation I


(2)  evaluate[[I:Identifier ]] = |give the value bound to id I
                                 or
                                 |give the value stored in the cell bound to id I

(3)  evaluate[[O:Operator(E1:Expression,E2:Expression)]] =
        ||evaluate E1 then give the value label #1
        |and evaluate E2 then give the value label #2
        then apply O

(4)  apply[[ADD]] = give sum(the value #1, the value #2)
```

Assume we want to translate the expression `x+1`. First we parse the expression and get the abstract syntax tree, e.g., ADD('x',1) . For better readability we omit type information. Now we unfold the function call evaluate⟦ADD('x',1)⟧ . By unfolding we mean that the call to a function is replaced by the according instance of the function definition.

```
evaluate⟦ADD('x',1)⟧ = | | | give the value bound to 'x'
                       | | | or give the value stored in the cell bound to id 'x'
                       | | then give the value label #1
                       | and
                       | | give 1 then give the value label #2
                       then give sum(the value#1, the value#2)
```

There are two alternatives in the or action, namely that `'x'` is bound to a cell or that it is directly bound to a value. Assume that the expression ADD('x',1) occurs in an actual program. Then sort analysis, as type checking of action terms is called, might detect that `'x'` is bound to a cell in the declaration part of the program. As a consequence the second alternative of or can be eliminated. Next this unfolded specification is translated to C using simple translation schemes like the following:

$$\begin{array}{lcl} [\![\mathit{give\ d\ label\ \#n}]\!] & \Rightarrow & d_i = [\![d]\!] \ \ (\mathit{associating\ n\ with\ } d_i) \\ [\![\mathit{the\ value\ \#n}]\!] & \Rightarrow & d_i \ \ (\mathit{where\ n\ is\ associated\ with\ } d_i) \\ [\![\mathit{the\ value\ stored\ in\ c}]\!] & \Rightarrow & \mathit{storage}[[\![c]\!]] \end{array}$$

Finally we get the following C code:

```
_d1 = storage[_BOUND("x",_b1).datum.cell];
_d2 = _MAKE_INTEGER(1); _d4 = _SUM(_d1,_d2);
```

3.11 Derivation of Abstract Machines

We use the term **abstract machine** to refer to an intermediate target language and a related target architecture for compilation. Using an abstract machine, compilation is done in two steps. First the source language program is compiled into a program in the abstract machine language. Next this abstract machine program is translated into a program in the target language or it can just be interpreted by an emulator (an interpreter for the abstract machine language). Abstract machines simplify the design of compilers and increase portability, since the first step of the compilation is independent of the final target language and the underlying hardware. This also facilitates code optimizations and native code generation.

Author	**Quality**	**Languages**	**Transformations**	**Papers**
Mitchell Wand	aut: no corr: no	spec: interpreter written in Scheme (continuation semantic) obj: abstract machine and translation functions src: a toy functional language	basically folding of common patterns into new instructions	[Wan86]
Peter Kursawe, Ulf Nilsson, Stephan Diehl	aut: in part ("Design Methodology") corr: no	spec: interpreter written in Prolog obj: abstract machine and translation schemes src: Prolog and TFS	partial deduction, folding and pass separation	[Kur87], [Nil93], [Nil92], [Die93]
John Hannan	aut: in part corr: yes	spec: natural semantics specification obj: abstract machine and translation schemes src: λ-calculus	pass separation, fold, unfold and specific transformations like closure introduction	[Han91a], [HM90], [Han91b], [Han94], [Ses95]
Ulrik Jørring, W.L. Scherlis	aut: transformations are not precisely defined corr: no	spec: functional program of an interpreter obj: functional programs of a compiler and an abstract machine src: a functional language	pass separation, partial evaluation, meta-transformation and several transformations on data structures	[JS86]

<u>**Comments:**</u>
In these approaches the target language (trg) is always the language of the derived abstract machine.
"Design Methodology" First Kursawe applied the methodology to Prolog's unification, then Nilsson to Prolog's control and finally the current author to an extension of Prolog, namely Typed Feature Structures.

Table 3.3: Approaches to derive abstract machines

As one of the challenging problems in partial evaluation and mixed computation, N.D. Jones mentions the derivation of abstract machines [10]. In the work shown in Table 3.3, there are three ways to derive abstract machines.

3.11.1 A Partial Evaluation-Based Approach

The first approach is based on the partial evaluation of example programs and folding of patterns in the resulting residual code. This approach was first used by Kursawe [Kur87] to derive some of the instructions of the Warren Abstract Machine years after its invention. Kursawe introduced special unification predicates and used these to make unification explicit in Prolog clauses. Then he partially evaluated these clauses with respect to the definition of the unification predicates. Next he replaced recurring patterns in the resulting partially evaluated clauses by "machine instructions". The following example shows this process for the simpe case of the clause $append([], L, L)$ [11].

$$
\begin{array}{l}
append([], L, L).\\
\quad\downarrow \text{ make unification explicit}\\
append(A1, A2, A3) \text{ :- } unipp([], A1), unipp(L, A2), unipp(L, A3).\\
\quad\downarrow \text{ partial evaluation}\\
append(A1, A2, A3) \text{ :- } (var(A1), A1 = []; [] == A1), unipp(A2, A3).\\
\quad\downarrow \text{ define 'machine instructions'}\\
append(A1, A2, A3) \text{ :- } unipp\_constant([], A1), unipp(A2, A3).
\end{array}
$$

and the new machine instruction:

[10] "Can a traditional run-time architecture be derived automatically from the interpreter text?" [Jon88] At the Dagstuhl Seminar on Partial Evaluation in February 1996 those challenging problems have been reviewed and the above problem was considered one of the still unsolved ones.

[11] The unification is made explicit by adding calls to a special predicate *unipp*. The definition of *unipp* is given below:

```
unipp(S,T) :- atomic(S), (var(T),T=S ; S==T).
unipp(S,T) :- var(S), S=T.
unipp(S,T) :- struct(S), ...
```

In addition to the steps shown here, Kursawe also made the heap representation of terms explicit. Partial evaluation basically unfolds the *unipp* predicate, i.e., a literal in the intermediate code is replaced by the corresponding instance of the body of one or more of its defining clauses. More precisely, let L be a literal and L_i :- G_i all variants of clauses for L. Then replace L by the disjunction $(G'_1, ..., G'_k)$, where $G'_i = (t_{01} = t_{i1}, ..., t_{0n} = t_{in}, G_i)$ and $L = p(t_{01}, ..., t_{0n})$ and $L_i = p(t_{i1}, ..., t_{in})$.

$$\boxed{unipp\_constant(C, A) \text{ :- } var(A), A = C; C == A.}$$

Note that the result of this method is twofold: (i) compilation of a program into machine code, (ii) design of an abstract machine by defining its machine instructions.

3.11.2 A Combinator-Based Approach

In the second approach proposed by Wand [12], one starts with semantic equations of a continuation semantics, i.e., a denotational semantics which models control flow using continuations instead of the direct style used in the introduction in Section 2.6. Conversion to continuation-passing style has already been discussed in Section 3.8.2.

$$\begin{array}{l}
\mathcal{E}[\![varname]\!] = \lambda env.\lambda cont.cont(env\ varname) \\
\mathcal{E}[\![exp_1 + exp_2]\!] = \\
\quad \lambda env.\lambda cont.\mathcal{E}[\![exp_1]\!]\ env\ (\lambda v_1.\mathcal{E}[\![exp_2]\!]\ env\ (\lambda v_2.cont(v_1 + v_2)))
\end{array}$$

These equations can be read as translation rules from the source language into λ-calculus. As in the previous approach, one looks for recurring patterns. Based on these patterns one defines machine instructions as combinators, i.e., λ-expressions without free variables. The motivations and insights which lead to the combinators listed below are far from being automatable.

$$\begin{array}{l}
fetch = \lambda varname.\lambda env.\lambda cont.cont(env\ varname) \\
seq_0 = \lambda e_1.\lambda e_2.\lambda env.\lambda cont.e_1\ env\ (e_2\ env\ cont) \\
seq_1 = \lambda e_1.\lambda e_2.\lambda env.\lambda cont.\lambda x.e_1\ env\ (e_2\ env\ cont\ x) \\
add = \lambda env.\lambda cont.\lambda v_1.\lambda v_2.cont\ (v_1 + v_2)
\end{array}$$

Next one folds the semantic equations with respect to these machine instructions:

$$\begin{array}{l}
\mathcal{E}[\![varname]\!] = fetch\ varname \\
\mathcal{E}[\![exp_1 + exp_2]\!] = seq_0(\mathcal{E}[\![exp_1]\!], seq_1(\mathcal{E}[\![exp_2]\!], add))
\end{array}$$

As a result, the semantic equations become translation rules from the source language into the abstract machine language. Actually if we apply these translation rules to a source language program, we get a combinator tree. Wand proves that $seq_k(seq_p(a, b), c) = seq_{k+p}(a, seq_k(b, c))$

[12] Wand was one of the first who dealt with the question of deriving abstract machines from the semantics of a language. In 1982 he proposed an approach based on combinators [Wan82]. To find suitable combinators was not automated and was a very difficult task. Actually the CAM was derived in a similar way [CCM85].

and that by this equivalence, the tree can be transformed into a linear representation with respect to the *seq* combinators. Here linearity means that the first argument to a *seq* combinator is never a *seq* combinator. We can think of the seq_k operator as a sequencing instruction which passes k values in registers to the next abstract machine instruction. For this linear representation, Wand derives rewrite rules which define the abstract machine:

$$\begin{array}{l}
\langle seq_0(fetch\ varname,\ p),\ env,\ cont,\ s\rangle \Rightarrow \langle p,\ env,\ cont,\ (env\ varname),\ s\rangle \\
\langle seq_1(fetch\ varname,\ p),\ env,\ cont,\ x,\ s\rangle \Rightarrow \langle p,\ env,\ cont,\ x,\ (env\ varname),\ s\rangle \\
\langle seq_0(add,\ p),\ env,\ cont,\ x_1,\ x_2,\ s\rangle \Rightarrow\ \langle p,\ env,\ cont,\ (x_1 + x_2),\ s\rangle \\
\langle seq_1(add,\ p),\ env,\ cont,\ x_1,\ x_2,\ x_3,\ s\rangle \Rightarrow\ \langle p,\ env,\ cont,\ x_1,\ (x_2 + x_3),\ s\rangle
\end{array}$$

A few comments are in order on these rewriting rules. Looking at the definitions of the combinators, we find that variables bound by λ have become components of the states in the above rewrite rules. An abstract machine program is a linear $seq_k(\iota, p)$ expression. Such a program is executed by performing ι and then executing the program rest p.

3.11.3 A Pass Separation-Based Approach

The third approach uses special pass separation transformations to split interpreters into compiling and executing parts, the latter being the abstract machine.
The following term rewriting rules (see Section 5.1.1 for a formal definition of term rewriting systems) define the transitions of an interpreter for the call-by-value λ-calculus for function application [13]. The rules rewrite states of the form $\langle C, L, S\rangle$, where C is a sequence of instructions, L is a list of environments, and S is a list of closures.

$$\boxed{\begin{array}{lll}
\langle apply(M,N); C, [E|L], S\rangle & \Rightarrow_I & \langle N; M; ap; C, [E|[E|L]], S\rangle \\
\langle ap; C, L, [clo(E, lam(M))|[V|S]]\rangle & \Rightarrow_I & \langle M; C, [(V,E)|L], S\rangle \\
\langle lambda(M); C, [E|L], S\rangle & \Rightarrow_I & \langle C, L, [clo(E, lam(M))|S]\rangle
\end{array}}$$

Hannan's pass separation [Han91a] transforms these term rewriting rules into the following compiler rules which translate an instruction of the source language into a sequence of abstract machine instructions

$$\boxed{\begin{array}{lll}
apply(M,N) & \Rightarrow_C & \overline{push}; N; M; ap \\
ap & \Rightarrow_C & \overline{app} \\
lambda(M) & \Rightarrow_C & \overline{lambda}(M)
\end{array}}$$

[13] The example is taken from [Han91a] where the interested reader can find the whole interpreter. Hannan didn't cope with cyclic bindings and **letrec** as we do for example in chapter 6.

and the following rules defining an abstract machine

$$
\begin{array}{|lll|}
\hline
\langle \overline{push}; C, [E|L], S\rangle & \Rightarrow_X & \langle C, [E|[E|L]], S\rangle \\
\langle \overline{app}; C, L, [clo(E, lam(C'))|[V|S]]\rangle & \Rightarrow_X & \langle C'; C, [(V, E)|L], S\rangle \\
\langle \overline{lambda}(C'); C, [E|L], S\rangle & \Rightarrow_X & \langle C, L, [clo(E, lam(C'))|S]\rangle \\
& & \\
\hline
\end{array}
$$

The compiler does only computations on program structures, whereas the abstract machine computes primarily on run-time structures. Let $\Rightarrow^*$ be the transitive closure of $\Rightarrow$. The relation of the compiler and abstract machine rules to the original interpreter rules is expressed by the following simplified implication (for the correct implication see Section 9.3.6), which says that executing the compiled program stops in an equivalent final state as the original program:

$$\text{If } \langle C, L, S\rangle \Rightarrow_I^* \langle nop, L', S'\rangle \text{ then } C \Rightarrow_C^* C'' \text{ and } \langle C'', L, S\rangle \Rightarrow_X^* \langle nop, L', S'\rangle.$$

In the first approach, it is difficult to guarantee completeness, since it is solely based on the analysis of examples. By completeness we mean that all source language programs can be compiled into code for the abstract machine. The second and third approach do not share this insufficiency.

3.12 Open Problems

Although there are lots of existing systems in SDCG, all of them are far from providing all the advantages listed in Section 3.3. Most of the systems have been results of one-man projects and never made it from the research prototype to a practical tool. There are still research problems which have not even been attacked.

AI methods Handwriting a compiler is a design process different from the methods used in compiler generators. Methods like those used in AI systems for engineering design (e.g., [TS92]) could be adapted and used in SDCG systems, thus leading us - in the pretentious terminology of AI - to knowledge-based semantics-directed compiler design systems.

Other Source Languages Existing SDCG systems do not address the questions of self-modification, concurrency, or parallelism in the source language.

Target Languages Almost all existing systems produce compilers for a fixed target language. In a real compiler generator, also the target language should be a parameter.

Abstract Machines A SDCG system could produce descriptions of abstract machines and a compiler, which produces code for this machine. Candidates for techniques used by such a generator have been discussed in Section 3.11. Note that none of these techniques was fully automated.

Other Specialization Methods There are other general specialization algorithms besides partial evaluation, which can be used for SDCG, e.g., deforestation (c.f. [Wad88]) or supercompilation (c.f. [Tur86],[SGJ94]). These specialization methods basically differ from partial evaluation in the amount and quality of information that is propagated, e.g., bindings of variables, equality and inequality of expressions.

Error Handling Generated compilers can only produce error messages in terms of their specification language. Additional specification language constructs might enable the generated compilers to use terms of the source language.

3.13 Conclusions

We classified existing SDCG systems according to several criteria. We discussed some of the criteria in more detail, namely the specification languages and transformations. We explained some of the methods used in the SDCG systems by means of examples. Although space did not permit to discuss the advantages and disadvantages of each of the systems, we discussed the general issues in Section 3.3 and 3.4. In Table 3.1 and 3.2 in the column **Transformations** almost all entries contain partial evaluation. In most SDCG systems partial evaluation has been extended by specific transformations often depending on the ideosyncrasies of the specification language transformed. Realistic compiler generators have been possible for specification languages with a rich set of basic operations. A closer look at the tables shows that most of the systems do not have correctness proofs, but correctness is one of the reasons why we want to use semantics specifications as a basis for compiler generation. Of great importance in the context of this thesis is that so far the derivation of abstract machines has not been automated, but only been assisted by transformation tools.

Chapter 4

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 [DM82], translations and optimizations [Mou93], static semantics and dynamics 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 thesis we use natural semantics to specify the dynamic semantics of programming languages. In this chapter we define our notation to write natural semantics specifications and give an example specification.

4.1 Introduction

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 [NN92]. This difference is also emphasized by the terms small-step and big-step semantics [Win93] or transitional style and relational style semantics [dS92]. At least in [NN92] 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 4.3.1). 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 it's 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 $\frac{\mathrm{P_2}\triangleright[[\mathrm{X}\mapsto \mathrm{E}^*]|\mathrm{E}]\rightarrow \mathrm{E}^* \quad \mathrm{P_1}\triangleright[[\mathrm{X}\mapsto \mathrm{E}^*]|\mathrm{E}]\rightarrow \mathrm{E}'}{\mathbf{letrec}\ \mathrm{X}=\mathrm{P_2}\ \mathbf{in}\ \mathrm{P_1}\ \mathbf{end}\triangleright \mathrm{E}\rightarrow \mathrm{E}'}$. 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 (see Section 2.4). Except of not allowing for cyclic dependencies, it can also be regarded as a notation to write natural semantics specifications.

4.2 Syntax

To begin with, 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 idea to split general and implementation details by the use of a separately given interpretation for function symbols [1] to ease compiler generation.

[1]We do not use the term Two-Level in the sense of [NN86, JGS93], where programs can be annotated to distinguish compile-time and run-time expressions, but as in in Peter Lee's High-Level semantics [Lee89]. In

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 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.

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
$\mathbf{R}$	$::=$	$\frac{\mathbf{J}^*}{\tilde{\mathbf{S}}}$	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).

Figure 4.1: Syntax of 2BIG

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 Figure 4.1. The following is an example of a 2BIG rule:

$$r = \frac{member((X \mapsto Y),S) \quad V \triangleright S \rightarrow N}{assign(X,V) \triangleright S \rightarrow replace(X,S,N)}$$

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.

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 5.3.2.
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.

4.3 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, dS92, dS90].

4.3.1 Inductive Systems

Definition 4.3.1 (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 4.3.2 (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 4.3.3 (Inductively Defined Set) *The* **set inductively defined by** *ϕ is defined as $\mathcal{I}(\phi) = \bigcap\{A | A \ is\ \phi - closed\}$.*

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

Theorem 4.3.4 *$\mathcal{I}(\phi)$ is ϕ-closed*

Proof. $P \subseteq \mathcal{I}(\phi) \overset{4.3.3}{\Longrightarrow}$ for every ϕ-closed set A: $P \in A \overset{4.3.2}{\Longrightarrow}$ for every ϕ-closed set A: $c \in A$ $\overset{4.3.3}{\Longrightarrow} c \in \mathcal{I}(\phi)$ □

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

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

Theorem 4.3.5 (Principle of Rule Induction) *Let Ψ be a predicate over U.*

$$\begin{array}{c}(\forall (P, c) \in \phi : (\forall x \in P : \Psi(x) \ is\ true) \Rightarrow \Psi(c) \ is\ true) \\ \Rightarrow \\ \forall a \in \mathcal{I}(\phi) : \Psi(a) \ is\ true\end{array}$$

An inductive system is **finitary** if the preconditions of all rules are finite.

Next we formally define a proof based on inductive rules as a sequence of items. Each item is an axiom or follows by a subset of items preceding that item in the sequence.

Definition 4.3.6 (Finite Length Proof) *Given a finitary inductive system ϕ, a sequence $\langle b_0, \ldots, b_n \rangle$ is a finite ϕ-proof of b if $b_n = b$ and for all $m \leq n$ there is a set $B \subseteq \{b_i : i < m\}$ such that $(B, b_m) \in \phi$.*

Theorem 4.3.7 (proof in [dS92]) *For every finitary inductive system ϕ we have that $\mathcal{I}(\phi) = \{b : \ b \text{ has a finite } \phi\text{-proof}\}$.*

$8 \in \mathcal{I}(\phi_1)$ 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 4.3.8 (Proof Tree) *Given a finitary inductive system ϕ, a ϕ-tree (or proof tree) of b, denoted $PT_\phi(b)$, is an object $\frac{PT_\phi(b_1)\ \dots\ PT_\phi(b_n)}{b}$ where there exists a rule $(\{b_1, \dots, b_n\}, b) \in \phi$, such that $PT_\phi(b_i)$ is a ϕ-tree for b_i and $(1 \leq i \leq n)$.*

Theorem 4.3.9 (proof in [dS92]) *For every finitary inductive system ϕ we have that $\mathcal{I}(\phi) = \{b : \ b \text{ has a finite } \phi\text{-tree}\}$.*

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

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

Theorem 4.3.10 (proof in [Ast91]) *Let $\Psi(X) = \{c | (P, c) \in \phi, P \subseteq X\}$. Then $\mathcal{I}(\phi)$ is the least fixed-point of Ψ.*

4.3.2 Relational Inductive Systems

Terms without variables are ground. The set of all **ground terms** over a signature [2] 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 4.3.11 (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.*

[2] For simplicity we do not consider many-sorted signatures as in [dS92] 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.

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 4.3.12 (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 $\sigma : F \to (({T_C}^0 \cup \ldots \cup {T_C}^m) \to (T_C \cup \{\perp\}))$ be an interpretation of the function names. 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. *$f \in F$*
   (a) *if $\forall i : \eta_\sigma(t_i) \neq \perp$ then $\eta_\sigma(t) = \sigma(f)(\eta_\sigma(t_1), \ldots, \eta_\sigma(t_n))$.*
   (b) *$\eta_\sigma(t) = \perp$*
2. *$f \in C$*
   (a) *if $\forall i : \eta_\sigma(t_i) \neq \perp$ then $\eta_\sigma(t) = f(\eta_\sigma(t_1), \ldots, \eta_\sigma(t_n))$.*
   (b) *$\eta_\sigma(t) = \perp$*

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 4.3.13 (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 $(\{p_1, \ldots, p_n\}, c) \in \phi$, θ is a substitution and $\theta(p_1), \ldots, \theta(p_n), \theta(c)$ 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: $\mathcal{I}(\overline{\phi_2}) = \mathcal{I}(\phi_1) = \{2^n | n \in \mathcal{N}, n > 0\}$.

4.4 Semantics of 2BIG

After transformation of side conditions into transitions (see Section 5.3.2), the 2BIG rules are simply relational inductive rules with ordered premises where we regard $\rightarrow$ 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 $c \triangleright e \rightarrow e' \in \mathcal{I}(\overline{\phi})$, which means that there is a $\overline{\phi}$-proof of $c \triangleright e \rightarrow e'$. 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 $c \triangleright e_1 \rightarrow e_2$ (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 .

4.5 Properties of 2BIG Rules

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

Definition 4.5.1 (Deterministic Rules) *A set ϕ of* 2BIG *rules is* **deterministic***, iff* $(B_1, c_1 \triangleright e_1 \rightarrow e'_1), (B_2, c_2 \triangleright e_2 \rightarrow e'_2) \in \phi \Rightarrow c_1, e_1$ *and* c_2, e_2 *are not unifiable.*

Definition 4.5.2 (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* c_1, e_1 *and* c_2, e_2 *are not unifiable or* $c_1, e_1 =_\alpha c_2, e_2$ *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 4.5.3 (Defining Occurrence) *An occurrence of a variable is* **defining***, iff it is on the left hand side of the conclusion or it is its first occurrence and this occurrence is on the right hand side of a precondition. All other occurrences are* **using***.*

Definition 4.5.4 (Well-Orderedness) *A rule is* **well-ordered***, iff every use of a variable is preceded by a defining occurrence of it.*

Definition 4.5.5 (Sequential Rules) *A set* ϕ *of* 2BIG *rules is* **sequential**, *iff* $(\{c_1 \triangleright e_1 \rightarrow e'_1, \ldots, c_m \triangleright e_m \rightarrow e'_m\}, c \triangleright e \rightarrow e') \in \phi \Rightarrow \forall 1 \leq i < m : e'_i = e_{i+1}$ *and* $e'_m = e'$.

Definition 4.5.6 (Temporary Variable) *A variable* x *is* **temporary** *in a rule* $(\{c_1 \triangleright e_1 \rightarrow e'_1, \ldots, c_n \triangleright e_n \rightarrow e'_n\}, c \triangleright e \rightarrow e')$, *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 4.5.7 (Allocated Rules) *A set* ϕ *of* 2BIG *rules is* **allocated**, *iff* $r \in \phi \Rightarrow$ *there is no temporary variable in* r.

4.6 Static Semantics

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 [dS90]. 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 chapter, 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.

4.7 Example: A 2BIG Specification

We consider a small fragment of a 2BIG specification of SIMP, just enough to construct a proof tree for the program **while i>1 do i:=i−1 od**.
First we specify how expressions are evaluated [3]. In this example the state will be a mapping of variable names to values. A transition of the form $E \triangleright S \rightarrow V$ means that in state S the expression E evaluates to V.

$$\frac{is\_id(X)}{X \triangleright S \rightarrow lookup(X,S)} \qquad \frac{is\_num(N)}{N \triangleright S \rightarrow N}$$

$$\frac{E_1 \triangleright S \rightarrow V_1 \quad E_2 \triangleright S \rightarrow V_2}{E_1 > E_2 \triangleright S \rightarrow greater(V_1,V_2)} \qquad \frac{E_1 \triangleright S \rightarrow V_1 \quad E_2 \triangleright S \rightarrow V_2}{E_1 - E_2 \triangleright S \rightarrow sub(V_1,V_2)}$$

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

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

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

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 X 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.
- $replace(X, S, V)$ 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.

[3] 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*:

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

And the following side-conditions:

- $is\_id(X)$ is *true* if X is an identifier.
- $is\_num(N)$ is *true* if N is a number.

Using the above 2BIG rules we can now construct a proof tree[4] 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**:

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

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

$$\frac{\dfrac{\mathbf{i} \triangleright [\mathbf{i}\mapsto 2] \rightarrow \text{V}_1 \qquad \mathbf{1} \triangleright [\mathbf{i}\mapsto \text{V}_2]}{\mathbf{i{>}1} \triangleright [\mathbf{i}\mapsto 2] \rightarrow \text{true} \equiv \text{greater}(\text{V}_1,\text{V}_2)} \qquad \dfrac{\mathbf{i{:=}i{-}1} \triangleright [\mathbf{i}\mapsto 2] \rightarrow \text{S}^* \qquad \mathbf{while\ \ i{>}1\ \ do\ \ i{:=}i{-}1\ \ od} \triangleright \text{S}^* \rightarrow \text{S}'}{\mathbf{i{:=}i{-}1;while\ \ i{>}1\ \ do\ \ i{:=}i{-}1\ \ od} \triangleright [\mathbf{i}\mapsto 2] \rightarrow \text{S}'}}{\mathbf{while\ \ i{>}1\ \ do\ \ i{:=}i{-}1\ \ od} \triangleright [\mathbf{i}\mapsto 2] \rightarrow \text{S}'}$$

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

$$\frac{\dfrac{\dfrac{\text{true}\boxed{1}}{\mathbf{i} \triangleright [\mathbf{i}\mapsto 2] \rightarrow 2\boxed{2}} \quad \dfrac{\text{true}\boxed{3}}{\mathbf{1} \triangleright [\mathbf{i}\mapsto 2] \rightarrow 1}}{\mathbf{i{>}1} \triangleright [\mathbf{i}\mapsto 2] \rightarrow \text{true}\boxed{4}} \quad \dfrac{\dfrac{\dfrac{\dfrac{\text{true}\boxed{5}}{\mathbf{i} \triangleright [\mathbf{i}\mapsto 2] \rightarrow 2\boxed{6}} \quad \dfrac{\text{true}\boxed{7}}{\mathbf{1} \triangleright [\mathbf{i}\mapsto 2] \rightarrow 1}}{\mathbf{i{-}1} \triangleright [\mathbf{i}\mapsto 2] \rightarrow 1\boxed{8}}}{\mathbf{i{:=}i{-}1} \triangleright [\mathbf{i}\mapsto 2] \rightarrow [\mathbf{i}\mapsto 1]\boxed{9}} \quad \dfrac{\dfrac{\dfrac{\text{true}\boxed{10}}{\mathbf{i} \triangleright [\mathbf{i}\mapsto 1] \rightarrow 1\boxed{11}} \quad \dfrac{\text{true}\boxed{12}}{\mathbf{1} \triangleright [\mathbf{i}\mapsto 1] \rightarrow 1}}{\mathbf{i{>}1} \triangleright [\mathbf{i}\mapsto 1] \rightarrow \text{false}\boxed{13}}}{\mathbf{while\ \ i{>}1\ \ do\ \ i{:=}i{-}1\ \ od} \triangleright [\mathbf{i}\mapsto 1] \rightarrow [\mathbf{i}\mapsto 1]}}{\mathbf{i{:=}i{-}1;while\ \ i{>}1\ \ do\ \ i{:=}i{-}1\ \ od} \triangleright [\mathbf{i}\mapsto 2] \rightarrow [\mathbf{i}\mapsto 1]}}{\mathbf{while\ \ i{>}1\ \ do\ \ i{:=}i{-}1\ \ od} \triangleright [\mathbf{i}\mapsto 2] \rightarrow [\mathbf{i}\mapsto 1]}$$

In the above proof, the results of evaluating functions and side conditions have been marked by boxed numbers $\ldots\boxed{n}$. Below we list these functions and side conditions.

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

[4]We formally defined proof trees in Section 4.3.1

Chapter 5

Generation of Compilers and Abstract Machines

We present a system that generates a compiler and an abstract machine from a Natural Semantics specification of a programming language. First an overview of the system and the transformations involved is given. There are three kinds of transformations, those transforming inference rules, the transformation from inference rules to term rewriting rules, and those transforming term rewriting rules. We define every transformation precisely and illustrate it by a simple example.

5.1 Introduction

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. By dividing compilation into two stages, abstract machines 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 [1]. In this thesis we distinguish

[1] There are three levels at which one might define an abstract machine:

- the way it is specified (see abstract interpreter in Section 5.3.10)
- the way it works (executes instructions and changes the state)
- the purpose it is used to (intermediate language for compilation).

mathematical machines and abstract machines, the latter being very much streamlined for compiler design. As an example we do not consider the Chemical Abstract Machine (a model for concurrency) to be an abstract machine in the latter sense. Common to most abstract machines seem to be a program store and a state, usually containing a stack.
For almost all kinds of languages, there exist abstract machines, e.g.:

Type of Language	Abstract Machines
imperative	P4 [PD82]
functional	SECD [Lan64], CAM [CCM85], FAM [Car84], G-Machine [Joh84]
logic	WAM [War77, AK91]
functional/logic	CAMEL [Müc92]
constraint	CLAM [JSMY92]
concurrent, constraint	OZAM [MSS95]
object oriented	[BFHW94], JavaVM [Mic95]*

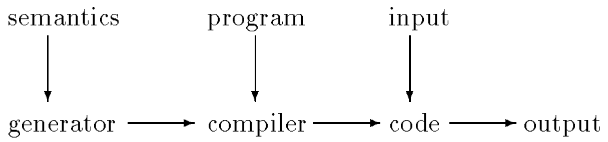
* In JAVA, a brand new language proposed for writing applications for the world wide web (WWW), sending abstract machine code over the network helps to cope with two new problems: the heterogenity of the network and the security (virus protection).

As discussed in Section 3.11, 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, Die93]. Another approach is to use pass separation transformations [JS86]. 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 into 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 determinate inductive rules. In McKeever's framework, the factorization algorithm of Fabio daSilva [dS90] 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

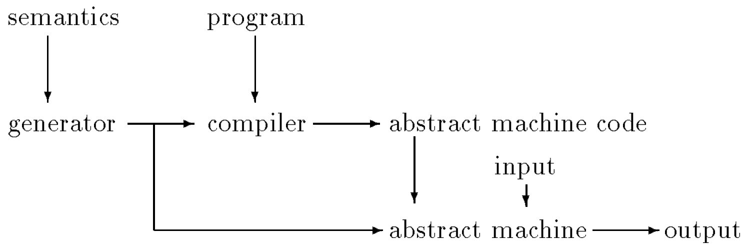
- the formal definition of a meta-language and the transformations,
- the implementation of the method,
- its application to the specifications of realistic programming languages

- and thus the detection of missing links (e.g., extension of factorization to more than two rules) and insufficiencies,
- the development of optimization transformations
- and the correctness proof of the core transformations.

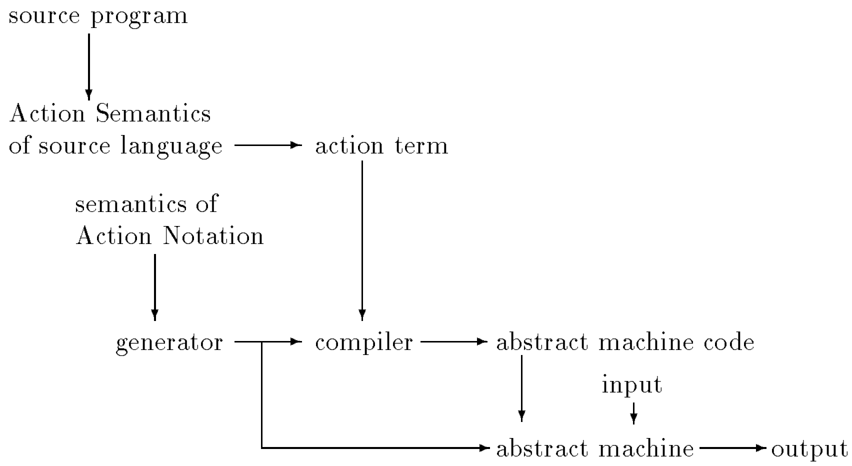
Given a semantics specification of a source language, current semantics-directed compiler generators produce compilers from the source language into a fixed target language.



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



We chose Action Notation 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 term rewriting systems.

5.1.1 Term Rewriting Systems

The following definitions are based on those given in [Han94, Jou95].

Definition 5.1.1 (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: $s(s(null))+s(null) \stackrel{1}{\Rightarrow}_{R_+} s(null)+s(s(null)) \stackrel{1}{\Rightarrow}_{R_+} null+ s(s(s(null))) \stackrel{1}{\Rightarrow}_{R_+} s(s(s(null)))$ and thus we have $s(s(null))+s(null) \stackrel{*}{\Rightarrow}_{R_+} s(s(s(null)))$.

5.1.2 Properties of Term Rewriting Systems

Definition 5.1.2 (Normal form, termination, confluence)
*A term t is in R-***normal form***, iff there exists no term u such that $t \Rightarrow_R u$. If $s \stackrel{*}{\Rightarrow}_R u$ and u is in normal form, then we write $s \stackrel{!}{\Rightarrow}_R u$. R is* **terminating** *iff there is no infinite sequence $t_1 \Rightarrow_R t_2 \Rightarrow_R \ldots$. R is* **confluent** *iff $(t \stackrel{*}{\Rightarrow}_R t_1$ and $t \stackrel{*}{\Rightarrow}_R t_2)$ implies that a term t' exists such that $t_1 \stackrel{*}{\Rightarrow}_R t'$ and $t_2 \stackrel{*}{\Rightarrow}_R t'$.*

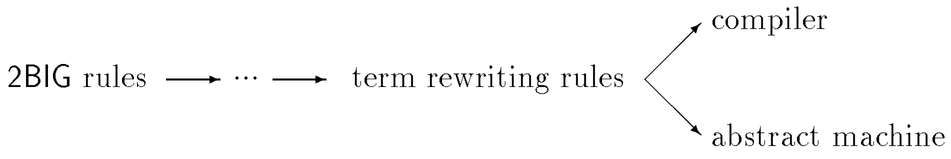
Definition 5.1.3 (Orthogonality) *R is* **overlapping** *if a left-hand side unifies with a renamed non-variable subterm of some other left-hand side or with a renamed proper subterm of itself. R is* **orthogonal** *if it is both linear and non-overlapping.*

Theorem 5.1.4 (proof in [Hue80]) *Every orthogonal system is confluent.*

In practice, functions (e.g., $+$, $-$, $\ldots$) 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 $s \stackrel{1}{\Rightarrow}_R \eta_\sigma(\theta(r))$ is a one-step reduction. The evaluation function η_σ was defined in Section 4.3.2.

5.2 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.

Original Rules

The 2BIG rules below define the evaluation of sum expressions[2]:

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

Proof tree for **add(num(1),add(num(2),num(3)))**

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

The result 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:

(i) After building the trees $\frac{}{\mathbf{num(2)}\triangleright nil \to 2}$ and $\frac{}{\mathbf{num(3)}\triangleright nil \to 3}$ 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 first proof tree, but is used again in the left hand side of the conclusion in the second proof tree.

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.

[2]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.

Allocation of Temporary Variables

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

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

Proof tree for **add(num(1),add(num(2),num(3)))**

$$\frac{\frac{}{\mathbf{num(1)} \triangleright [[[\mathrm{nil}]],\mathrm{nil}] \to [[[\mathrm{nil}]],1]} \qquad \frac{\frac{}{\mathbf{num(2)} \triangleright [[[\mathrm{nil}],[1]],\mathrm{nil}] \to [[[\mathrm{nil}],[1]],2]\boxed{1}} \qquad \frac{}{\mathbf{num(3)} \triangleright [[[2],[1]],\mathrm{nil}]\boxed{2} \to [[[2],[1]],3]\boxed{3}}}{\mathbf{add(num(2),num(3))} \triangleright [[[1]],\mathrm{nil}] \to [[[1]],5]\boxed{4}}}{\mathbf{add(num(1),add(num(2),num(3)))} \triangleright [[],\mathrm{nil}] \to [[],6]}$$

When we build the new proof tree from left to right we first build $\frac{}{\mathbf{num(2)} \triangleright [[[nil],[1]],nil] \to [[[nil],[1]],2]}$ and then $\frac{}{\mathbf{num(3)} \triangleright [[[2],[1]],nil] \to [[[2],[1]],3]}$. It turns out that the two observations we made before do no longer hold:

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

ad (ii): *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 $t_1 \stackrel{1}{\Rightarrow} \ldots \stackrel{1}{\Rightarrow} t_n$. Here the term t_{i+1} which 'results' from the rewriting step $t_i \stackrel{1}{\Rightarrow} t_{i+1}$ is the 'input' to the next rewriting step $t_{i+1} \stackrel{1}{\Rightarrow} t_{i+2}$.

Sequentialization 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.

$$\frac{E_1 \triangleright [[[S]|D],S] \rightarrow [[[S]|D],V_1] \qquad conv_1 \triangleright [[[S]|D],V_1] \rightarrow [[[V_1]|D],S] \qquad E_2 \triangleright [[[V_1]|D],S] \rightarrow [[[V_1]|D],V_2] \qquad conv_2 \triangleright [[[V_1]|D],V_2] \rightarrow [D,plus(V_1,V_2)]}{\mathbf{add}(\mathrm{E}_1,\mathrm{E}_2)\triangleright[\mathrm{D},\mathrm{S}]\rightarrow[\mathrm{D},\mathrm{plus}(\mathrm{V}_1,\mathrm{V}_2)]}$$

$$\frac{}{\mathbf{num}(N)\triangleright[D,S]\rightarrow[D,N]} \qquad \frac{}{conv_1\triangleright[D,V]\rightarrow[[[V]|D],S]} \qquad \frac{}{conv_2\triangleright[[[V_1]|D],V_2]\rightarrow[D,plus(V_1,V_2)]}$$

Proof tree for **add(num(1),add(num(2),num(3)))**

$$\frac{
\begin{array}{c}\overline{\mathbf{num}(\mathbf{1})}\\ \triangledown \\ [[[nil]],nil] \\ \downarrow \\ [[[nil]],1]\end{array}
\quad
\begin{array}{c}\overline{conv_1}\\ \triangledown \\ [[[nil]],1] \\ \downarrow \\ [[[1]],nil]\end{array}
\quad
\begin{array}{c}
\dfrac{
\begin{array}{c}\overline{\mathbf{num}(\mathbf{2})}\\ \triangledown \\ [[[nil],[1]],nil] \\ \downarrow \\ [[[nil],[1]],2]\boxed{1}\end{array}
\begin{array}{c}\overline{conv_1}\\ \triangledown \\ [[[nil],[1]],2]\boxed{1'} \\ \downarrow \\ [[[2],[1]],nil]\boxed{2'}\end{array}
\begin{array}{c}\overline{\mathbf{num}(\mathbf{3})}\\ \triangledown \\ [[[2],[1]],nil]\boxed{2} \\ \downarrow \\ [[[2],[1]],3]\boxed{3}\end{array}
\begin{array}{c}\overline{conv_2}\\ \triangledown \\ [[[2],[1]],3]\boxed{3'} \\ \downarrow \\ [[[1]],5]\boxed{4'}\end{array}
}{\mathbf{add}(\mathbf{num}(\mathbf{2}),\mathbf{num}(\mathbf{3}))} \\ \triangledown \\ [[[1]],nil] \\ \downarrow \\ [[[1]],5]\boxed{4}
\end{array}
\quad
\begin{array}{c}\overline{conv_2}\\ \triangledown \\ [[[1]],5] \\ \downarrow \\ [[],6]\end{array}
}{\mathbf{add}(\mathbf{num}(\mathbf{1}),\mathbf{add}(\mathbf{num}(\mathbf{2}),\mathbf{num}(\mathbf{3})))\triangleright[[],nil]\rightarrow[[],6]}$$

Returning to the observations for our previous proof tree we find:

ad (iii): The state $\boxed{1}$ is equal to $\boxed{1'}$ which is converted to $\boxed{2'}$ and $\boxed{2'}$ is equal to $\boxed{2}$.

ad (iv): The state $\boxed{3}$ is equal to $\boxed{3'}$ which is converted to $\boxed{4'}$ and $\boxed{4'}$ is equal to $\boxed{4}$.

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

Conversion into Term Rewriting System

In the generated 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 $\mathbf{add}(E_1, E_2)$. 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.

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

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 rules defining a compiler

$$\begin{array}{lll}
\mathbf{num}(N) & \Rightarrow_C & \overline{num}(N) \\
\mathbf{add}(E_1, E_2) & \Rightarrow_C & \overline{add}; E_1; \overline{conv}_1; E_2; \overline{conv}_2
\end{array}$$

and the following rules defining an abstract machine

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

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.

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

Note, that this rewriting sequence has the same length as the rewriting sequence we got 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 which has to be executed. After execution of the instruction, the program counter is increased by one.

5.3 Generating Compilers and Abstract Machines from 2BIG Specifications

An overview of the system is given in Figure 5.1. 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 lead 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:

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

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

5.3.1 Source Variables

Compile-time objects are those which can be constructed or evaluated at compile-time on grounds of the program without having to refer to information in the state or input to the program. Because of the halting problem, not all compile-time objects can be computed or constructed at compile-time. Thus one has to find approximations, i.e., criteria which help to detect as many compile-time objects as possible.

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 $C;\textbf{while } B \textbf{ do } C \textbf{ od} \triangleright S \rightarrow S'$, are also known at compile-time. In the proof tree for $\textbf{while i} > \textbf{1 do i} := \textbf{i} - \textbf{1 od} \triangleright [\textbf{i} \mapsto 2] \rightarrow [\textbf{i} \mapsto 1]$ at the end of Section 4.7, there is no term on the left of $\triangleright$, which was not present in the original program $\textbf{while i} > \textbf{1 do i} := \textbf{i} - \textbf{1 od}$.

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

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

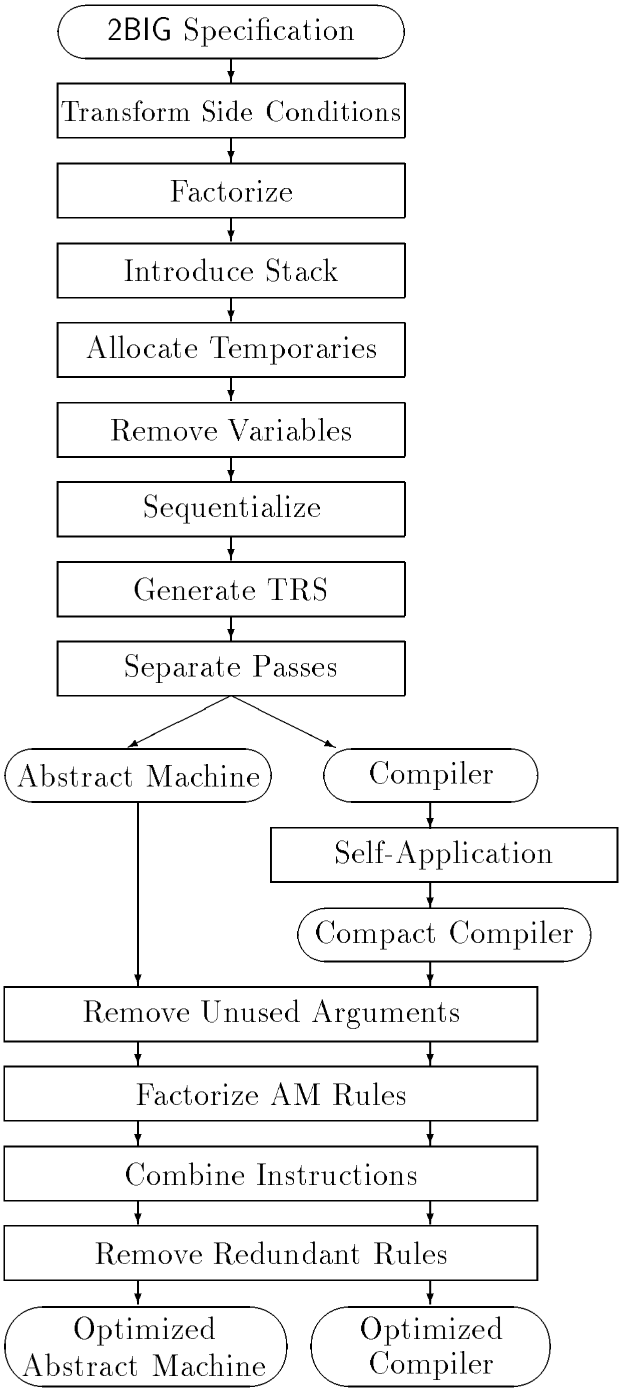


Figure 5.1: Overview of the Transformations

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. Our translation 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.

5.3.2 Transformation of Side Conditions

As we mentioned before, we regard side conditions as syntactic sugar for a special kind 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 the characteristic function of a predicate, side conditions can be converted into transitions. Let $\frac{s_1 \ \cdots \ s_j}{c \triangleright e \rightarrow e'}$ 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

$$a(x_1, \ldots, x_k) \triangleright [y_1, \ldots, y_m] \rightarrow true$$

A side condition of the form *not* $p(t_1, \ldots, t_n)$ is converted into a transition

$$a(x_1, \ldots, x_k) \triangleright [y_1, \ldots, y_m] \rightarrow false$$

where a is a new instruction symbol, for which we generate a new rule

$$\frac{}{a(x_1,\ldots,x_k)\triangleright[y_1,\ldots,y_m]\rightarrow\chi_p(t_1,\ldots,t_n)}$$

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 $\mathbf{bool}(B)$ we get

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

5.3.3 Factorization

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 [AU72, ASU86, LP81] 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):

> Given the rules $A \rightarrow \alpha\beta_1|\ldots|\alpha\beta_n|\gamma_1|\ldots|\gamma_m$ for the nonterminal A.
> Grammar rules for the same nonterminal which have a common prefix α are replaced by a single rule $A \rightarrow \alpha A'$ and a new nonterminal A' is introduced which produces the suffixes $\beta_1,\ldots,\beta_n$. Thus we get the new rules $A \rightarrow \alpha A'|\gamma_1|\ldots|\gamma_m$ and $A' \rightarrow \beta_1|\ldots|\beta_n$. 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.

Factorization of inference rules converts sets of determinate inductive rules into deterministic rules. The following factorization transformation can be regarded as an extension of da Silva's transformation[3] 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 factorization until there are no more conflicting rules.
By $=_\alpha$ we will denote equality of terms and formulae modulo renaming of variables.

Definition 5.3.2 (Conflicting Rules) *Two rules are* **conflicting**, *if they have the same left hand sides in their conclusions.*

Let $\mathcal{C}$ 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 non-terminal)*:

$$\frac{c_{11} \triangleright e_{11} \rightarrow e'_{11} \quad \dots \quad c_{1m_1} \triangleright e_{1m_1} \rightarrow e'_{1m_1}}{c_1 \triangleright e_1 \rightarrow e'_1} \quad \dots \quad \frac{c_{n1} \triangleright e_{n1} \rightarrow e'_{n1} \quad \dots \quad c_{nm_n} \triangleright e_{nm_n} \rightarrow e'_{nm_n}}{c_n \triangleright e_n \rightarrow e'_n}$$

where $c_1, e_1 =_\alpha \dots =_\alpha c_n, e_n$.
Let θ be a renaming of variables and j be the largest integer, such that for all $p, q \in \{1, \dots, n\}$: $(c_{pj}, e_{pj})\theta = (c_{qj}, e_{qj})\theta$ and

$$\forall k < j : (c_{pk}, e_{pk}, e'_{pk})\theta = (c_{qk}, e_{qk}, e'_{qk})\theta \tag{5.1}$$

It should be kept in mind that by the renaming θ the terms are α-equal and not only unifiable. 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)*
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:

[3] A factorization transformation has been defined and proved correct by daSilva [dS90] for the case of sets with two conflicting rules. A motivating example for our extension is given in Appendix D.

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

where x is a new variable name.

The common term is the most general term modulo renaming of variables which unifies with both e_1 and e_2.

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

$$\begin{array}{l} \text{Variables used in the remaining preconditions:} \\ \quad \mathcal{R}_1 = \bigcup_{k=1}^{n} \mathcal{V}(c_{k(j+1)},\ldots,c_{km_k},e_{k(j+1)},\ldots,e_{km_k},e'_k)\theta \\ \text{Variables defined in } seg\text{, the conclusion or in } c_{kj}, e_{kj}\text{:} \\ \quad \mathcal{R}_2 = \mathcal{V}(seg,c_1,e_1)\theta \cup \bigcup_{k=1}^{n} \mathcal{V}(c_{kj},e_{kj})\theta \\ \text{Variables used in the remaining preconditions, defined in } seg\text{, etc.} \\ \text{and not passed in the common term } e^\bullet\text{:} \\ \quad \mathcal{R}_3 = (\mathcal{R}_1 \cap \mathcal{R}_2) - \mathcal{V}(e^\bullet)\theta \\ \text{Those variables in } \mathcal{R}_3 \text{ which are not source variables} \\ \quad \mathcal{R} = \mathcal{R}_3 - \mathcal{V}(\theta(c_1)) \\ \text{Source variables are passed as arguments} \\ \quad \kappa = \iota(x_1,\ldots,x_m) \quad \textit{where } x_i \in \mathcal{R}_3 \cap \mathcal{V}(\theta(c_1)) \end{array} \tag{5.2}$$

(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.)

Let e' be a new variable name, then $\mathcal{C}$ is replaced by:

$$\frac{seg \quad c_{1j} \triangleright e_{1j} \rightarrow e^\bullet \quad \kappa \triangleright [\mathcal{R}, e^\bullet] \rightarrow e'}{c_1 \triangleright e_1 \rightarrow e'} \theta$$

$$\frac{c_{1(j+1)} \triangleright e_{1(j+1)} \rightarrow e'_{1(j+1)} \quad \cdots \quad c_{1m_1} \triangleright e_{1m_1} \rightarrow e'_{1m_1}}{\kappa \triangleright [\mathcal{R}, e'_{1j}] \rightarrow e'_1} \theta$$

$$\vdots$$

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

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

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

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

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

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

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

5.3.4 Stack Introduction

In the next step, the states in the rules are extended by a stack[4]. This stack will be used later to store temporary variables[5]. A rule of the form

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

is converted into

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

[4]In our examples we use the variable D for the stack or **d**ump, because S is already used for the store.

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

where s is a new variable name.

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

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

5.3.5 Allocation of Temporary Variables

Intuitively 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. More precisely:

Definition 5.3.3 (Temporary in a Precondition) *Let* $\frac{c_1 \rhd e_1 \rightarrow e'_1 \quad \ldots \quad c_n \rhd e_n \rightarrow e'_n}{c \rhd e \rightarrow e'}$ *be a* 2BIG *rule, then we define the sets* $\mathcal{M}_j$ $(1 \leq j \leq n)$ *of variables temporary in the precondition with index* j*:*

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

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

It can easily be shown that this definition of temporary variables is equivalent to Definition 4.5.6.
Now consider the rules resulting from stack introduction:

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

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 factorization would destroy common initial segments. Consider the two 2BIG rules for $\mathbf{bool}(B)$ 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:

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

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**.

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

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

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

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

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

5.3.6 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[6]. We say that source and input state variables (see Def. 5.3.1) 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 $\frac{c_1 \triangleright e_1 \rightarrow e'_1 \ \ldots \ c_n \triangleright e_n \rightarrow e'_n}{c \triangleright e \rightarrow e'}$ be a rule. For all $i > 1$ we define:

$$\mathcal{M}_i = \{x : x \in \mathcal{V}(c_i) \ and \ x \notin \mathcal{V}(c, e)\}.$$

For every $\mathcal{M}_i \neq \emptyset$ a new instruction p_i is introduced and the rule is transformed into

$$\frac{c_1 \triangleright e_1 \rightarrow e'_1 \ \ldots \ c_{i-1} \triangleright e_{i-1} \rightarrow e'_{i-1} \ \ p_i \triangleright e'_{i-1} \rightarrow e'_i \ \ c_{i+1} \triangleright e_{i+1} \rightarrow e'_{i+1} \ \ldots \ c_n \triangleright e_n \rightarrow e'_n}{c \triangleright e \rightarrow e'}$$

The new instructions are defined by the following rules:

$$\frac{c_i \triangleright e_i \rightarrow e'_i}{p_i \triangleright e'_{i-1} \rightarrow e'_i}$$

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

[6] In McKeever's work the form of rules has been much more restrictive:

$$\frac{c_1 \triangleright e_1 \rightarrow e'_1 \ \ldots \ c_n \triangleright e_n \rightarrow e'_n}{a(p_1, \ldots, p_m) \triangleright e \rightarrow e'}$$

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:

$$\frac{\textbf{lookup}(X) \triangleright S \rightarrow E \qquad E \triangleright S \rightarrow S'}{\textbf{call}(X) \triangleright S \rightarrow S'}$$

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

$$\frac{\textbf{lookup}(X) \triangleright [[[S]|D],S] \rightarrow [[[S]|D],E] \qquad E \triangleright [D,S] \rightarrow [D,S']}{\textbf{call}(X) \triangleright [D,S] \rightarrow [D,S']}$$

Now the above transformation can be applied:

$$\frac{\textbf{lookup}(X) \triangleright [[[S]|D],S] \rightarrow [[[S]|D],E] \qquad exec \triangleright [[[S]|D],E] \rightarrow [D,S']}{\textbf{call}(X) \triangleright [D,S] \rightarrow [D,S']} \qquad \frac{E \triangleright [D,S] \rightarrow [D,S']}{exec \triangleright [[[S]|D],E] \rightarrow [D,S']}$$

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

5.3.7 Sequentialization

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

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

is transformed into

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

and we add the rules

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

where for each rule of the form $p_i \triangleright e_i' \rightarrow e_{i+1}$, the instruction p_i has the form $\iota(x_1, \ldots, x_k)$, ι is a new instruction symbol and $\{x_1, \ldots, x_k\} = (\mathcal{V}(e_{i+1}) - \mathcal{V}(e_i')) \cap \mathcal{V}(c_0)$. One might ask, why we do not add a precondition $p_0 \triangleright e_0 \rightarrow e_1$? 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 $\mathcal{V}(e_{i+1}) - \mathcal{V}(e'_i)$.

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

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

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

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

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

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

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

5.3.8 Conversion into Term Rewriting Systems

After sequentialization, 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 $c \triangleright e \rightarrow e'$ 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 $\frac{c_1 \triangleright e_1 \rightarrow e_1' \quad \ldots \quad c_n \triangleright e_n \rightarrow e_n'}{c \triangleright e \rightarrow e'}$ are converted into $\langle (c; p), e\rangle \rightarrow \langle (c_1; \ldots; c_n; p), e_1\rangle$ where p is a new variable name.

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

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

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 (see proofs in Section 9.3.5). By Theorem 5.1.4 we conclude that these term rewriting systems are confluent.

5.3.9 Pass Separation

The term **staging transformation** has been introduced in [JS86] 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 $p_{\overline{x}}(y) = p(\overline{x}, y)$. [7] In contrast, **pass separation** transforms the program p into two programs p_1 and p_2 such that $p_2(p_1(x), y) = p(x, y)$. [8] What is important about this equation is that here p_1 produces some intermediate data, which are input to p_2. 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

[7] There is a trivial solution to this equation, namely $p_{\overline{x}} = \lambda y.(\lambda x.p(x,y)\ \overline{x})$. The goal is to move computations from p to $p_{\overline{x}}$.

[8] There is a trivial solution to this equation, namely $p_1 = id$ and $p_2 = p$. The goal is to move computations from p_2 to p_1.

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 somewhat hand-waving article [JS86] and the provably correct pass separation transformations for term rewriting systems [Han94] and evolving algebras [Die95b].

5.3.10 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 5.3.4 (Term Complexity) *The complexity of a term t is defined by structural induction:*

- $\mathcal{C}(\iota) = 1$ *for a constant* ι;
- $\mathcal{C}(X) = 1$ *for a variable* X;
- $\mathcal{C}(\iota(t_1, \ldots, t_n)) = \mathcal{C}(t_1) + \cdots + \mathcal{C}(t_n) + 1$.

From this we derive a partial order on terms: $t_1 \sqsubset t_2$ *iff* $\mathcal{C}(t_1) < \mathcal{C}(t_2)$.

Definition 5.3.5 (Abstract Interpreter (Def. 4.1 in [Han94]))
(Σ, R) *is an* **abstract interpreter** *iff*

- (Σ, R) *is a linear term rewriting system*
- *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 above for our running example is an abstract interpreter.

Definition 5.3.6 (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 5.3.7 (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 5.3.8 (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. $\mathcal{V}(b') \subseteq \{X_1, \ldots, X_k\}$
2. $b_i \sqsubset \iota(X_1, \ldots, X_k)$*, for all* $1 \leq i \leq n$
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 5.3.1), (2) ensures that the compiler will be normalizing, i.e., compilation will terminate, (3) is needed to show confluence of the generated compiler (by Theorem 5.1.4) 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 5.3.9 (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 for each*
   $\langle \iota(X_1, \ldots, X_k); C, e \rangle \Rightarrow \langle a_1; \ldots; a_m; b'; C, e' \rangle \in R|_{\iota,k}$, such that $b' =$ $\mathsf{suffix}(R|_{\iota,k})$ *and*
   $\iota(X_1, \ldots, X_k) \Rightarrow_c \kappa(X_1, \ldots, X_k); b' \in R_c$,
   $a_1; \ldots; a_m \overset{*}{\Rightarrow}_{R_c} a_c, \quad e \overset{*}{\Rightarrow}_{R_c} e_c, \quad e' \overset{*}{\Rightarrow}_{R_c} e'_c$
   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 the following holds:

$$\text{if } \langle c, e\rangle \overset{*}{\Rightarrow}_R \langle c', e'\rangle \text{ then } c \overset{*}{\Rightarrow}_{R_c} \tilde{c}\ ,\ e \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e}\ ,\ c' \overset{\dagger}{\Rightarrow}_{R_c} \tilde{c}'\ ,\ e' \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e}' \textit{ and } \langle \tilde{c}, \tilde{e}\rangle \overset{*}{\Rightarrow}_{R_x} \langle \tilde{c}', \tilde{e}'\rangle$$

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.

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 $\langle ... \rangle$), i.e., they never apply to sub-terms of that term, because we do not allow $\langle ... \rangle$ to occur in a term. As a result they can be implemented more efficiently than ordinary rewrite rules.

Pass separating the term rewriting system for our example results in the following compiler rules

$$\begin{array}{lcl}
\textbf{while } B \textbf{ do } C \textbf{ od} & \Rightarrow & \overline{while}(B, C) \\
\textbf{bool}(B) & \Rightarrow & \overline{bool}(B); B; conv_2; test; conv_3; factor_2 \\
conv_2 & \Rightarrow & \overline{conv}_2 \\
conv_3 & \Rightarrow & \overline{conv}_3 \\
factor_2 & \Rightarrow & \overline{factor}_2 \\
test & \Rightarrow & \overline{test}
\end{array}$$

And the following executor rules are generated:

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

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 at runtime into $B; \overline{conv}; \overline{factor}(C, B)$. Looking at the definition of the pass separation transformation, we find that $B; \overline{conv}; \overline{factor}(C, B)$ was a candidate for b', but $\overline{factor}(C, B) \not\sqsubset$ **while** B **do** C **od**, i.e., $\overline{factor}(C, B)$ is as complex regarding number and structure of the arguments as the original **while** instruction. Hannan only allows compiler rules which translate an instruction into a sequence of less complex instructions [9]. Using this restriction he can prove that the generated compiler is strongly normalizing, i.e., terminates for all source language programs.

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

5.4 Optimizations of Compiler and Abstract Machine

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

5.4.1 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, i.e., for those instructions introduced by the transformations.

> For example the 5 compiler rules for **bool** can be reduced to one rule
>
> $$\mathbf{bool}(B) \Longrightarrow \overline{bool}(B); B; \overline{conv}_2; \overline{test}; \overline{conv}_3; \overline{factor}_2$$

[9] Restriction 2 of definition 5.3.8.

5.4.2 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

$$\mathbf{bool}(B) \Longrightarrow \overline{bool}; B; \overline{conv_2}; \overline{test}; \overline{conv_3}; \overline{factor}_2$$

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.

5.4.3 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 factorize rules dealing with error handling. Often the error handling was identical for several instructions.

Here is a simple example:

$$
\begin{array}{lll}
\langle hd; C, [H|T]\rangle & \Longrightarrow & \langle C, H\rangle \\
\langle hd; C, error\rangle & \Longrightarrow & \langle C, error\rangle \\
\langle tl; C, [H|T]\rangle & \Longrightarrow & \langle C, T\rangle \\
\langle tl; C, error\rangle & \Longrightarrow & \langle C, error\rangle
\end{array}
$$

The error case can be factored out:

$$
\begin{array}{lll}
\langle fact(hd); C, [H|T]\rangle & \Longrightarrow & \langle C, H\rangle \\
\langle fact(tl); C, [H|T]\rangle & \Longrightarrow & \langle C, T\rangle \\
\langle fact(X); C, error\rangle & \Longrightarrow & \langle C, error\rangle
\end{array}
$$

In general the factorization transformation for abstract machine rules works as follows. Let k_1 and k_2 be two instructions with the same number of arguments. Let R_1 be the set of all rules for k_1 and R_2 be the set of all rules for k_2. The intersection of two rule sets modulo instruction names is defined as:

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

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

$$
\begin{array}{rl}
R_1 \boxed{-} R_2 = \{r_1 : & r_1 \in R_1\ and\ there\ exists\ no\ rule\ in\ R_2 \\
& and\ no\ variable\ renaming\ as\ in\ the\ definition\ of\ \boxed{\cap}\}
\end{array}
$$

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 $\langle \iota(k_1, x_1, \ldots, x_n); c,\ e\rangle \Rightarrow s \in \widehat{R}$.
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 $R_1 \cup R_2$ by $\widehat{R}$.

5.4.4 Combining Instructions

Transformation of side conditions and sequentialization 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 [10]

$$\langle i_j; \ldots; i_{j+k}, e_j \rangle \overset{*}{\hookrightarrow}_{R_x} \langle nop, e' \rangle$$

then we define a new instruction. 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:

$$\langle \iota(x_1, \ldots, x_p); C, e_j \rangle \Longrightarrow \langle C; e' \rangle$$

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

As an example look at the compiler rule for $\mathbf{bool}(B)$. The sequence $\overline{conv}_2; \overline{test}; \overline{conv}_3;$ is replaced by a new instruction $\overline{comb}$:

$$\mathbf{bool}(B) \Longrightarrow \overline{bool}; B; \overline{comb}; \overline{factor}_2$$

The new instruction replaces the three original ones:

$$\langle \overline{comb}; P, [D, V] \rangle \Longrightarrow \langle P, [D, [[V], is\_bool(V)]] \rangle$$

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

5.4.5 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 factorization 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.

That these optimizations greatly reduce the number of compiler and abstract machine rules is pointed out in Section 7.6 for the action notation specification. First, by self-application, the number of compiler rules was reduced from 216 to 43. Second, using the other optimizations we got 181 instead of 276 abstract machine rules.

Chapter 6

Generation of a Compiler and Abstract Machine for Mini-ML

We apply our system to a specification of Mini-ML . The generated compiler and abstract machine are similar to those presented in [CCM85], i.e., the Categorial Abstract Machine (CAM). The CAM has been the basis of very efficient implementations of ML [Ler93, MS86]. Based on the specification of Mini-ML in [Kah87, Des86] we will present a 2BIG specification of Mini-ML and the compiler and abstract machine generated by our system. A closer look at the abstract machine instructions reveals that variable lookup is still inefficient. We introduce an abstract syntax and a conversion of Mini-ML programs into the abstract syntax. In the abstract syntax variable names are replaced by access paths which are just a different encoding for deBruijn numerals. For this abstract syntax we give a 2BIG specification. Now it turns out that the generated abstract machine is close to the CAM. Whereas in this chapter the emphasis is on how modifications in the input specification help to improve the generated compiler and abstract machine, in the next chapter we will trace the generation of an abstract machine.

6.1 Mini-ML

In [Kah87, Des86] the authors present natural semantics specifications of Mini-ML, the CAM and the translation of Mini-ML programs to CAM code. Mini-ML consists of the purely applicative part of ML, more precisely a simple typed λ-calculus with constants, pairs, conditionals and recursive function definitions . The syntax of Mini-ML is given in Figure 6.1, its semantics will be defined in the next section by 2BIG rules.

x		variable symbol	
n		number	
b		boolean value: $true, false$	
E	::=	**bool**(b) | **num**(n)	boolean value, number
	|	**equal**(E, E)	equality test
	|	$E + E$ | $E - E$	sum, difference
	|	**var**(x)	variable use
	|	**if** E **then** E **else** E **end**	conditional
	|	(E, E)	pair
	|	**fst**(E)	first value of pair
	|	**snd**(E)	second value of pair
	|	$\lambda x.E$	abstraction
	|	$(E\ E)$	function application
	|	**let** $x = E$ **in** E **end**	function definition
	|	**letrec** $x = E$ **in** E **end**	recursive function definition

Figure 6.1: Syntax of Mini-ML

The following example program counts from 10 down to 0:

$$\begin{array}{ll}\textbf{letrec } y = \lambda x.\textbf{if } \textbf{equal}(\textbf{var}(x), \textbf{num}(0)) & \textbf{then } \textbf{var}(x) \\ & \textbf{else } (\textbf{var}(y)\ (\textbf{var}(x) - \textbf{num}(1))) \\ \textbf{in } (\textbf{var}(y)\ \textbf{num}(10))\ \textbf{end} & \end{array}$$

And this is a Mini-ML function computing Fibonacci numbers:

$$\begin{array}{l}\textbf{letrec } fib = \lambda x.\textbf{if } \textbf{equal}(\textbf{var}(x), \textbf{num}(0))\ \textbf{then } \textbf{num}(0) \\ \qquad\qquad \textbf{else if } \textbf{equal}(\textbf{var}(x), \textbf{num}(1))\ \textbf{then } \textbf{num}(1) \\ \qquad\qquad\qquad \textbf{else } (\textbf{var}(fib)\ (\textbf{var}(x) - \textbf{num}(1))) \\ \qquad\qquad\qquad\qquad +(\textbf{var}(fib)\ (\textbf{var}(x) - \textbf{num}(2))) \\ \qquad\qquad\qquad \textbf{end} \\ \qquad\qquad \textbf{end} \\ \textbf{in } (\textbf{var}(fib)\ \textbf{num}(10))\ \textbf{end}\end{array}$$

6.2 Transforming a 2BIG specification of Mini-ML

6.2.1 Cyclic Bindings

Recall from Chapter 4, that we require well-orderedness. Well-orderedness does not allow for cyclic dependencies of variables and thus allows us to use terms and not graphs or rational trees as the underlying model of 2BIG rules. From a practical point of view facilitates to generate term rewriting systems as specifications of abstract machines . To implement cyclic dependencies as for E^* in $\frac{\mathrm{P}_2 \triangleright [[\mathrm{X} \mapsto \mathrm{E}^*] | \mathrm{E}] \to \mathrm{E}^* \quad \mathrm{P}_1 \triangleright [[\mathrm{X} \mapsto \mathrm{E}^*] | \mathrm{E}] \to \mathrm{E}'}{\textbf{letrec } \mathrm{X} = \mathrm{P}_2 \textbf{ in } \mathrm{P}_1 \textbf{ end} \triangleright \mathrm{E} \to \mathrm{E}'}$ we lift the handling of cyclic bindings of variables in the meta-language into the specification, i.e., we specify the indirection and dereferencing of variables by 2BIG rules. As a result the state in the 2BIG rules contains a new component, namely a list of redirections. Redirections associate indices (addresses) with values. Everytime a value is an index, we have to look up its value in the redirections. This process is also called dereferencing.

6.2.2 Basic Operations

In the 2BIG rules of this chapter the following functions are used:

- $lookup(E, X)$ yields the value associated with the identifier bound to X in the mapping E.
- $replace(X, V, E)$ yields a new mapping, which differs from E only in that the association of the identifier X is replaced by an association of the identifier X to the value V.
- $minus\_op(V_1, V_2)$ yields $V_1 - V_2$ if both values are numbers
- $plus\_op(V_1, V_2)$ yields $V_1 + V_2$ if both values are numbers
- $equal\_op(V_1, V_2)$ yields $true$ if both values are equal, $false$ otherwise.
- $lookup\_red(R, N)$ yields the value associated with index N in the redirections R.
- $replace\_red(N, V, R)$ replaces the value associated with index N in the redirections R by the value V.
- $new\_index(R)$ yields a new index, which is not yet contained in R.

6.2.3 First Shot

The following 2BIG specification is based on the natural semantics specification of Mini-ML in [Kah87, Des86]. To convert their natural semantics rules into 2BIG rules we had to remove the cyclic dependencies and enforce the static semantics of 2BIG .

2BIG Specification of Mini-ML

We use constructors like **xbool**, **xnum** and **clo** to build intermediate data structures and distinguish them from constructors of the source language like **bool** or **num**. The constructor **val** indicates that a value has not to be dereferenced, whereas **ind** is an index and has to be dereferenced to get a value.

Primitive Types

$$\frac{}{\mathbf{num}(N) \triangleright [R, E] \rightarrow [R, \mathbf{xnum}(N)]} \tag{6.1}$$

$$\frac{V_1 \triangleright [R, E] \rightarrow [R', \mathbf{xnum}(N)] \qquad V_2 \triangleright [R', E] \rightarrow [R'', \mathbf{xnum}(M)]}{V_1 + V_2 \triangleright [R, E] \rightarrow [R'', \mathbf{xnum}(plus\_op(N, M))]} \tag{6.2}$$

$$\frac{V_1 \triangleright [R, E] \rightarrow [R', \mathbf{xnum}(N)] \qquad V_2 \triangleright [R', E] \rightarrow [R'', \mathbf{xnum}(M)]}{V_1 - V_2 \triangleright [R, E] \rightarrow [R'', \mathbf{xnum}(minus\_op(N, M))]} \tag{6.3}$$

$$\frac{}{\mathbf{bool}(B) \triangleright [R, E] \rightarrow [R, \mathbf{xbool}(B)]} \tag{6.4}$$

$$\frac{N \triangleright [R, E] \rightarrow [R', N'] \qquad M \triangleright [R', E] \rightarrow [R'', M']}{\mathbf{equal}(N, M) \triangleright [R, E] \rightarrow [R'', \mathbf{xbool}(equal\_op(N', M'))]} \tag{6.5}$$

Pairs

$$\frac{V_1 \triangleright [R, E] \rightarrow [R', V_1'] \qquad V_2 \triangleright [R', E] \rightarrow [R'', V_2']}{(V_1, V_2) \triangleright [R, E] \rightarrow [R'', \mathbf{xpair}(V_1', V_2')]} \tag{6.6}$$

$$\frac{V \triangleright [R, E] \rightarrow [R', \mathbf{xpair}(A, B)]}{\mathbf{fst}(V) \triangleright [R, E] \rightarrow [R', A]} \qquad \frac{V \triangleright [R, E] \rightarrow [R', \mathbf{xpair}(A, B)]}{\mathbf{snd}(V) \triangleright [R, E] \rightarrow [R', B]} \tag{6.7}$$

Variable Lookup

$$\frac{\mathbf{lkup} \triangleright [X, E] \rightarrow \mathbf{ind}(N)}{\mathbf{var}(X) \triangleright [R, E] \rightarrow [R, lookup\_red(R, N)]} \qquad \frac{\mathbf{lkup} \triangleright [X, E] \rightarrow \mathbf{val}(V)}{\mathbf{var}(X) \triangleright [R, E] \rightarrow [R, V]} \tag{6.8}$$

$$\frac{}{\mathbf{lkup} \triangleright [X, E] \rightarrow lookup(E, X)} \tag{6.9}$$

Conditional

$$\frac{B \triangleright [R, E] \rightarrow [R', \mathbf{xbool}(\mathbf{true})] \qquad V_1 \triangleright [R', E] \rightarrow [R'', V_1']}{\mathbf{if}\ B\ \mathbf{then}\ V_1\ \mathbf{else}\ V_2\ \mathbf{end} \triangleright [R, E] \rightarrow [R'', V_1']} \tag{6.10}$$

$$\frac{B \triangleright [R, E] \rightarrow [R', \mathbf{xbool}(\mathbf{false})] \qquad V_2 \triangleright [R', E] \rightarrow [R'', V_2']}{\mathbf{if}\ B\ \mathbf{then}\ V_1\ \mathbf{else}\ V_2\ \mathbf{end} \triangleright [R, E] \rightarrow [R'', V_2']} \tag{6.11}$$

Functions

$$\frac{}{\lambda X.V \triangleright [R,E] \rightarrow [R, \mathbf{clo}(E, \mathbf{xlambda}(X,V))]} \tag{6.12}$$

$$\frac{V_1 \triangleright [R,E] \rightarrow [R', \mathbf{clo}(E', \mathbf{xlambda}(X,C))] \quad V_2 \triangleright [R',E] \rightarrow [R'',V_2'] \quad \mathbf{run} \triangleright [C, R'', replace(X, \mathbf{val}(V_2'), E')] \rightarrow [R^*, V]}{(V_1\ V_2) \triangleright [R,E] \rightarrow [R^*,V]} \tag{6.13}$$

$$\frac{C \triangleright [R,E] \rightarrow [R',V]}{\mathbf{run} \triangleright [C,R,E] \rightarrow [R',V]} \tag{6.14}$$

$$\frac{V_1 \triangleright [R,E] \rightarrow [R',V_1'] \quad V_2 \triangleright [R', replace(X, \mathbf{val}(V_1'), E)] \rightarrow [R'',V]}{\mathbf{let}\ X = V_1\ \mathbf{in}\ V_2\ \mathbf{end} \triangleright [R,E] \rightarrow [R'',V]} \tag{6.15}$$

Recursive Functions

$$\frac{\mathbf{newind} \triangleright R \rightarrow N \quad V_1 \triangleright [[\mathbf{red}(N, \mathbf{ind}(N))|R], replace(X, \mathbf{ind}(N), E)] \rightarrow [R',V_1'] \quad V_2 \triangleright [replace\_red(N, V_1', R'), replace(X, \mathbf{val}(V_1'), E)] \rightarrow [R'',V_2']}{\mathbf{letrec}\ X = V_1\ \mathbf{in}\ V_2\ \mathbf{end} \triangleright [R,E] \rightarrow [R'',V_2']} \tag{6.16}$$

$$\frac{}{\mathbf{newind} \triangleright R \rightarrow new\_index(R)} \tag{6.17}$$

Generated Compiler for Mini-ML

Applying our system to the above specification we get the following compiler rules:

$\mathbf{bool}(B)$	$\Rightarrow$	$\overline{\mathbf{bool}}(B)$
$\mathbf{equal}(A, B)$	$\Rightarrow$	$\overline{\mathbf{equal}}; A; \overline{\mathbf{conv\_0}}; B; \overline{\mathbf{conv\_1}}$
$\mathbf{num}(N)$	$\Rightarrow$	$\overline{\mathbf{num}}(N)$
$A + B$	$\Rightarrow$	$\overline{\mathbf{equal}}; A; \overline{\mathbf{conv\_2}}; B; \overline{\mathbf{conv\_3}}$
$A - B$	$\Rightarrow$	$\overline{\mathbf{equal}}; A; \overline{\mathbf{conv\_2}}; B; \overline{\mathbf{conv\_5}}$
(A, B)	$\Rightarrow$	$\overline{\mathbf{equal}}; A; \overline{\mathbf{conv\_0}}; B; \overline{\mathbf{conv\_9}}$
$\mathbf{fst}(V)$	$\Rightarrow$	$\overline{\mathbf{fst}}; V; \overline{\mathbf{conv\_10}}$
$\mathbf{snd}(V)$	$\Rightarrow$	$\overline{\mathbf{fst}}; V; \overline{\mathbf{conv\_11}}$
$\mathbf{var}(X)$	$\Rightarrow$	$\overline{\mathbf{var}}(X)$
$\mathbf{if}\ B\ \mathbf{then}\ V_1\ \mathbf{else}\ V_2\ \mathbf{end}$	$\Rightarrow$	$\overline{\mathbf{equal}}; B; \overline{\mathbf{conv\_7}}; \overline{\mathbf{factor\_1}}(V_2, V_1)$
$\lambda X.V$	$\Rightarrow$	$\overline{\mathbf{lambda}}(X, V)$
$(V_1\ V_2)$	$\Rightarrow$	$\overline{\mathbf{equal}}; V_1; \overline{\mathbf{conv\_12}}; V_2; \overline{\mathbf{conv\_13}}; \overline{\mathbf{run}}$
$\mathbf{let}\ X = V_1\ \mathbf{in}\ V_2\ \mathbf{end}$	$\Rightarrow$	$\overline{\mathbf{equal}}; V_1; \overline{\mathbf{conv\_15}}(X); V_2$
$\mathbf{letrec}\ X = V_1\ \mathbf{in}\ V_2\ \mathbf{end}$	$\Rightarrow$	$\overline{\mathbf{letrec}}; \overline{\mathbf{newind}}; \overline{\mathbf{conv\_16}}(X); V_1; \overline{\mathbf{conv\_17}}; V_2$

The names of the instructions introduced by our system are $\overline{\mathbf{test}}$_..., $\overline{\mathbf{conv}}$_..., $\overline{\mathbf{factor}}$_..., $\overline{\mathbf{fact}}$_... and $\overline{\mathbf{comb}}$_.... These names describe the task of an instruction not as specific as those instruction names we know from existing abstract machines. After a closer inspection of what our generated instructions do, we could give them more suggestive names like *push*, *pop*, *branch*, etc.

Generated Abstract Machine for Mini-ML

We give only some of the generated abstract machine rules here, to illustrate and discuss some inefficiencies in the machine.

Variable Lookup

$\langle \overline{\mathbf{var}}(X); C,\ [D, [R, E]]\rangle$	$\Rightarrow$	$\langle \overline{\mathbf{lkup}}; \overline{\mathbf{conv\_6}}; \overline{\mathbf{factor\_0}}; C,\ [[[R]\vert D], [X, E]]\rangle$
$\langle \overline{\mathbf{lkup}}; C,\ [D, [X, E]]\rangle$	$\Rightarrow$	$\langle C,\ [D, lookup(X, E)]\rangle$
$\langle \overline{\mathbf{conv\_6}}; C,\ [[[R]\vert D], S]\rangle$	$\Rightarrow$	$\langle C,\ [D, [[R], S]]\rangle$
$\langle \overline{\mathbf{factor\_0}}; C,\ [D, [[R], \mathbf{ind}(N)]]\rangle$	$\Rightarrow$	$\langle C,\ [D, [R, lookup\_red(R, N)]]\rangle$
$\langle \overline{\mathbf{factor\_0}}; C,\ [D, [[R], \mathbf{val}(V)]]\rangle$	$\Rightarrow$	$\langle C,\ [D, [R, V]]\rangle$

In the CAM the variable access is done by statically compiled access paths. In the above abstract machine rules variable access is still a search process hidden in the basic operation *lookup*. The instruction $\overline{\mathbf{factor\_0}}$ tests the cases that the variable is bound to a redirection or directly to a value.

Recursive Functions

$$
\begin{array}{l}
\langle\overline{\mathbf{letrec}}; C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [[[R,E]|D],R]\rangle \\
\langle\overline{\mathbf{newind}}; C,\ [D,R]\rangle \Rightarrow \langle C,\ [D, new\_index(R)]\rangle \\
\langle\overline{\mathbf{conv}}\mathbf{\_16}(X); C,\ [[[R,E]|D],N]\rangle \Rightarrow \\
\quad \langle C,\ [[[N,X,E]|D],[[\mathbf{red}(N,\mathbf{ind}(N))|R], replace(X,\mathbf{ind}(N),E)]]\rangle \\
\langle\overline{\mathbf{conv}}\mathbf{\_17}; C,\ [[[N,X,E]|D],[R,V]]\rangle \Rightarrow \\
\quad \langle C,\ [D,[replace\_red(N,V,R), replace(X,\mathbf{val}(V),E)]]\rangle
\end{array}
$$

Again the search process is hidden in a basic operation, namely $replace(X, \mathbf{ind}(N), E)$ and $replace(X, \mathbf{val}(V), E)$. The association found for the variable X is then replaced by an association of X to the new value. In the CAM the environment is implemented as a stack and the new value is just put on top of the stack.

6.2.4 Second Shot

As pointed out the structure of environments and as a result the variable lookup in the above abstract machine is still insufficient. Our transformations do not automatically change this structure, thus the 2BIG specification has to be modified. We introduce an abstract syntax with deBruijn numerals and a conversion of Mini-ML programs into the abstract syntax. Then we give a new 2BIG specification for Mini-ML programs in abstract syntax. When we replace variable names by deBruijn numerals the environment can be replaced by a stack. The value associated with the variable represented by the numeral \$0 is the top most element of the stack, for the numeral $\$n$ the value is the $(n-1)$th element of the stack. Access paths are a unary representation of numerals, i.e., the numeral \$0 is represented by **car**, the numeral $\$n$ by $\underbrace{\mathbf{cdr}(\mathbf{cdr}(\ldots\mathbf{cdr}}_{n\ times}(\mathbf{car})))$.

Abstract Syntax

Since we use now deBruijn numerals in λ,**let** and **letrec** abstractions, there are no more variable names in Mini-ML programs and we have to change the syntax (see Figure 6.2).

V	::=	$car \mid cdr(V)$	variable access path
E	::=	...	
	$\mid$	V	variable use
	$\mid$	$\lambda.E$	abstraction
	$\mid$	**let** E **in** E **end**	function definition
	$\mid$	**letrec** E **in** E **end**	recursive function definition

Figure 6.2: Abstract Syntax of Mini-ML

Using the new abstract syntax the example program which counts from 10 down to 0 becomes:

$$\begin{array}{ll}\textbf{letrec } \lambda.\textbf{if equal}(\textbf{car}, \textbf{num}(0)) & \textbf{then car} \\ & \textbf{else } (\textbf{cdr}(\textbf{car})\ (\textbf{car} - \textbf{num}(1))) \\ \textbf{in } (\textbf{car num}(10)) \textbf{ end} & \end{array}$$

And the Mini-ML function for Fibonacci numbers is now written as:

$$\begin{array}{l}\textbf{letrec } \lambda.\textbf{if equal}(\textbf{car}, \textbf{num}(0)) \textbf{ then num}(0) \\ \qquad\qquad \textbf{else if equal}(\textbf{car}, \textbf{num}(1)) \textbf{ then num}(1) \\ \qquad\qquad\qquad\quad \textbf{else } (\textbf{cdr}(\textbf{car})\ (\textbf{car} - \textbf{num}(1))) \\ \qquad\qquad\qquad\qquad +(\textbf{cdr}(\textbf{car})\ (\textbf{car} - \textbf{num}(2))) \\ \qquad\qquad\qquad \textbf{end} \\ \qquad\qquad \textbf{end} \\ \textbf{in } (\textbf{car num}(10)) \textbf{ end}\end{array}$$

Conversion into Abstract Syntax

$$\frac{}{\textbf{var}(X) \triangleright E \rightarrow access\_path(X, E)} \tag{6.18}$$

Here $access\_path(X, E)$ is a basic operation which yields the access path for the numeral \$$(n-1)$ if X is the n-th variable in E. Or in other words $access\_path(X, E) = \underbrace{\textbf{cdr}(\textbf{cdr}(\ldots\textbf{cdr}}_{n-1\ times}(\textbf{car})))$.

$$\frac{V \triangleright [X|E] \rightarrow V'}{\lambda X.V \triangleright E \rightarrow \lambda.V'} \tag{6.19}$$

$$\frac{V_1 \triangleright E \rightarrow V_1' \quad V_2 \triangleright [X|E] \rightarrow V_2'}{\textbf{let } X = V_1 \textbf{ in } V_2 \textbf{ end} \triangleright E \rightarrow \textbf{let } V_1' \textbf{ in } V_2' \textbf{ end}} \tag{6.20}$$

$$\frac{V_1 \triangleright [X|E] \rightarrow V_1' \quad V_2 \triangleright [X|E] \rightarrow V_2'}{\textbf{letrec } X = V_1 \textbf{ in } V_2 \textbf{ end} \triangleright E \rightarrow \textbf{letrec } V_1' \textbf{ in } V_2' \textbf{ end}} \tag{6.21}$$

In all other cases the arguments are just translated in the current environment, e.g.:

$$\frac{V_1 \triangleright E \rightarrow V_1' \quad V_2 \triangleright E \rightarrow V_2'}{(V_1, V_2) \triangleright E \rightarrow (V_1', V_2')} \tag{6.22}$$

2BIG Specification of Mini-ML

Variable Lookup Instead of rules for $\mathbf{var}(X)$, we have now rules for access paths:

$$\frac{}{\textbf{car} \triangleright [R, [\textbf{ind}(M)|E]] \rightarrow [R, \mathit{lookup\_red}(R, M)]} \tag{6.23}$$

$$\frac{}{\textbf{car} \triangleright [R, [\textbf{val}(V)|E]] \rightarrow [R, V]} \tag{6.24}$$

$$\frac{A \triangleright [R, E] \rightarrow [R, V]}{\textbf{cdr}(A) \triangleright [R, [H|E]] \rightarrow [R, V]} \tag{6.25}$$

Functions The name of the variable bound by the λ abstraction is no longer stored in the closure.

$$\frac{}{\lambda.C \triangleright [R, E] \rightarrow [R, \textbf{clo}(E, \textbf{xlambda}(C))]} \tag{6.26}$$

Instead of replacing the value bound to the variable bound by the λ abstraction of the closure, we now pass the new value on top of the environment, which is now a stack and no longer a mapping of variable names to values.

$$\frac{\begin{array}{l} V_1 \triangleright [R, E] \rightarrow [R', \textbf{clo}(E', \textbf{xlambda}(C))] \qquad V_2 \triangleright [R', E] \rightarrow [R'', V_2'] \\ \textbf{run} \triangleright [C, R'', [\textbf{val}(V_2')|E']] \rightarrow [R^*, V] \end{array}}{(V_1\ V_2) \triangleright [R, E] \rightarrow [R^*, V]} \tag{6.27}$$

Again, instead of binding the value V_1' to a variable it is passed on top of the stack.

$$\frac{V_1 \triangleright [R, E] \rightarrow [R', V_1'] \qquad V_2 \triangleright [R', [\textbf{val}(V_1')|E]] \rightarrow [R'', V_2']}{\textbf{let } V_1 \textbf{ in } V_2 \textbf{ end} \triangleright [R, E] \rightarrow [R'', V_2']} \tag{6.28}$$

Recursive Functions And again, the value of V_1' is now passed on top of the stack.

$$\frac{\mathbf{newind} \triangleright R \rightarrow M \qquad V_1 \triangleright [[\mathbf{red}(M, \mathbf{ind}(M))|R], [\mathbf{ind}(M)|E]] \rightarrow [R', V_1'] \quad V_2 \triangleright [replace\_red(M, V_1', R'), [\mathbf{val}(V_1')|E]] \rightarrow [R'', V_2']}{\mathbf{letrec}\ V_1\ \mathbf{in}\ V_2\ \mathbf{end} \triangleright [R, E] \rightarrow [R'', V_2']} \tag{6.29}$$

Generated Compiler for Mini-ML

Access paths are now translated into sequences of instructions, thus **cdr(cdr(car))** becomes $\overline{\mathbf{cdr}};\overline{\mathbf{cdr}};\overline{\mathbf{car}}$. Of course, there are no more abstract machine instructions, which take variable names as their arguments, e.g., X in $\overline{\mathbf{lambda}}(X, C)$.

car	$\Rightarrow$	$\overline{\mathbf{car}}$
$\mathbf{cdr}(A)$	$\Rightarrow$	$\overline{\mathbf{cdr}}; A$
$\lambda.C$	$\Rightarrow$	$\overline{\mathbf{lambda}}(C)$
$(V_1\ V_2)$	$\Rightarrow$	$\overline{\mathbf{equal}}; V_1; \overline{\mathbf{conv\_11}}; V_2; \overline{\mathbf{conv\_12}}; \overline{\mathbf{run}}$
let V_1 **in** V_2 **end**	$\Rightarrow$	$\overline{\mathbf{equal}}; V_1; \overline{\mathbf{conv\_14}}; V_2$
letrec V_1 **in** V_2 **end**	$\Rightarrow$	$\overline{\mathbf{letrec}}; \overline{\mathbf{newind}}; \overline{\mathbf{conv\_15}}; V_1; \overline{\mathbf{conv\_16}}; V_2$

Generated Abstract Machine for Mini-ML

Now we give the definitions generated for the instructions appearing in the above compiler rules.

Variable Lookup

$\langle\overline{\mathbf{car}}; C,\ [D, [R, [\mathbf{ind}(M)|E]]]\rangle$	$\Rightarrow$	$\langle C,\ [D, [R, lookup\_red(R, M)]]\rangle$
$\langle\overline{\mathbf{car}}; C,\ [D, [R, [\mathbf{val}(V)|E]]]\rangle$	$\Rightarrow$	$\langle C,\ [D, [R, V]]\rangle$
$\langle\overline{\mathbf{cdr}}; C,\ [D, [R, [V|E]]]\rangle$	$\Rightarrow$	$\langle C,\ [D, [R, E]]\rangle$

Functions

$$
\begin{array}{|l|}
\hline
\langle \overline{\mathbf{lambda}}(V); C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [D,[R,\mathbf{clo}(E,\mathbf{xlambda}(V))]]\rangle \\
\langle \overline{\mathbf{equal}}; C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [[[E]|D],[R,E]]\rangle \\
\langle \overline{\mathbf{conv\_11}}; C,\ [[[E]|D],[R,\mathbf{clo}(E',\mathbf{xlambda}(T))]]\rangle \Rightarrow \\
\quad \langle C,\ [[[E',T]|D],[R,E]]\rangle \\
\langle \overline{\mathbf{conv\_12}}; C,\ [[[E,T]|D],[R,V]]\rangle \Rightarrow \langle C,\ [D,[T,R,[\mathbf{val}(V)|E]]]\rangle \\
\langle \overline{\mathbf{run}}; C,\ [D,[T,R,E]]\rangle \Rightarrow \langle T; \overline{\mathbf{conv\_13}}; C,\ [[[T]|D],[R,E]]\rangle \\
\langle \overline{\mathbf{conv\_13}}; C,\ [[[T]|D],[R,E]]\rangle \Rightarrow \langle C,\ [D,[R,E]]\rangle \\
\langle \overline{\mathbf{conv\_14}}; C,\ [[[E]|D],[R,V]]\rangle \Rightarrow \langle C,\ [D,[R,[\mathbf{val}(V)|E]]]\rangle \\
\hline
\end{array}
$$

Recursive Functions

$$
\begin{array}{|l|}
\hline
\langle \overline{\mathbf{letrec}}; C,\ [D,[R,E]]\rangle \Rightarrow \langle C,\ [[[R,E]|D],R]\rangle \\
\langle \overline{\mathbf{conv\_15}}; C,\ [[[R,E]|D],M]\rangle \Rightarrow \\
\quad \langle C,\ [[[M,E]|D],[[\mathbf{red}(M,\mathbf{ind}(M))|R],[\mathbf{ind}(M)|E]]]\rangle \\
\langle \overline{\mathbf{conv\_16}}; C,\ [[[M,E]|D],[R,V]]\rangle \Rightarrow \\
\quad \langle C,\ [D,[\mathit{replace\_red}(M,V,R),[\mathbf{val}(V)|E]]]\rangle \\
\langle \overline{\mathbf{newind}}; C,\ [D,R]\rangle \Rightarrow \langle C,\ [D,\mathit{new\_index}(R)]\rangle \\
\hline
\end{array}
$$

6.2.5 Comparison to CAM

Before we can compare our generated abstract machine to the CAM, we have to introduce the CAM in more detail. We restrict the presentation to how variable lookup, λ-abstraction, function application and recursive function definitions are translated to CAM code and discuss the relevant instructions of the CAM. We will use the specifications in [Des86] [1], where both the CAM and the translation to CAM code are given by natural semantics rules. To avoid having to introduce another notation, we will write these rules in 2BIG notation, although they do not satisfy the static semantics of 2BIG.

Translation of Mini-ML to CAM Code

In our generated compiler, we have two stages. First the Mini-ML program is converted into the abstract syntax with access paths. This stage was described by inference rules. Then the actually generated compiler is described by TRS rules and translates a program in abstract

[1] Despeyroux's specification slightly differs from [Kah87] and [CCM85]. In the latter the CAM is specified by TRS rules and an ML program is given, which translates Mini-ML programs to CAM code.

syntax into an abstract machine program. These two stages have been intertwined in the inference rules below which specify a translation of Mini-ML programs into CAM code.

In these translation rules the state is an environment, in fact a list of variable names and we use $(\ldots,\ldots)$ to construct such lists.

$$\frac{\mathbf{access}(\mathrm{X}) \triangleright \mathrm{E} \rightarrow \mathrm{C}}{\mathbf{var}(\mathrm{X}) \triangleright \mathrm{E} \rightarrow \mathrm{C}}$$

Here **access**(X) computes given the environment E the access path for the variable X as a sequence of **car** and **cdr** instructions.

$$\frac{\mathrm{V} \triangleright \mathrm{E} \rightarrow \mathrm{C}}{\lambda \mathrm{X}.\mathrm{V} \triangleright \mathrm{E} \rightarrow \mathbf{cur}(\mathrm{C})}$$

The variable name is ignored and the expression V is translated into CAM code and used as an argument to the CAM instruction **cur**.

$$\frac{\mathrm{V}_1 \triangleright \mathrm{E} \rightarrow \mathrm{C}_1 \qquad \mathrm{V}_2 \triangleright \mathrm{E} \rightarrow \mathrm{C}_2}{(\mathrm{V}_1\ \mathrm{V}_2) \triangleright \mathrm{E} \rightarrow \mathbf{push};\mathrm{C}_1;\mathbf{swap};\mathrm{C}_2;\mathbf{cons};\mathbf{app}}$$

Here V_1 and V_2 are translated first and the resulting code is combined into a sequence of instructions.

$$\frac{\mathrm{V}_1 \triangleright (\mathrm{E},\mathrm{X}) \rightarrow \mathrm{C}_1 \qquad \mathrm{V}_2 \triangleright (\mathrm{E},\mathrm{X}) \rightarrow \mathrm{C}_2}{\mathbf{letrec}\ \mathrm{X}{=}\mathrm{V}_1\ \mathbf{in}\ \mathrm{V}_2\ \mathbf{end} \triangleright \mathrm{E} \rightarrow \mathbf{push};\mathbf{quote}(\mathrm{R}');\mathbf{cons};\mathbf{push};\mathrm{C}_1;\mathbf{swap};\mathbf{rplac};\mathrm{C}_2}$$

Note, that R' is an anonymous variable and, as we will see now when we present the CAM instructions, R' is used by **rplac** to create a cyclic binding.

Instructions of the CAM

In the rules for the instructions of the CAM the state is a stack of environments.

$$\frac{}{\mathbf{car} \triangleright [(\mathrm{A},\mathrm{B})|\mathrm{S}] \rightarrow [\mathrm{A}|\mathrm{S}]} \qquad \frac{}{\mathbf{cdr} \triangleright [(\mathrm{A},\mathrm{B})|\mathrm{S}] \rightarrow [\mathrm{B}|\mathrm{S}]}$$

$$\frac{}{\mathbf{push} \triangleright [\mathrm{A}|\mathrm{S}] \rightarrow [\mathrm{A}|[\mathrm{A}|\mathrm{S}]]} \qquad \frac{}{\mathbf{swap} \triangleright [\mathrm{A}|[\mathrm{B}|\mathrm{S}]] \rightarrow [\mathrm{B}|[\mathrm{A}|\mathrm{S}]]}$$

$$\frac{}{\mathbf{cons} \triangleright [\mathrm{A}|[\mathrm{B}|\mathrm{S}]] \rightarrow [(\mathrm{A},\mathrm{B})|\mathrm{S}]} \qquad \frac{\mathrm{C} \triangleright [(\mathrm{R},\mathrm{A})|\mathrm{S}] \rightarrow \mathrm{S}_1}{\mathbf{app} \triangleright [(\mathrm{clo}(\mathrm{C},\mathrm{R}),\mathrm{A})|\mathrm{S}] \rightarrow \mathrm{S}_1}$$

$$\frac{}{\mathbf{quote}(\mathrm{R}) \triangleright [\mathrm{A}|\mathrm{S}] \rightarrow [\mathrm{R}|\mathrm{S}]} \qquad \frac{\mathrm{V}{=}\mathrm{R}'}{\mathbf{rplac} \triangleright [(\mathrm{R},\mathrm{V})|[\mathrm{R}'|\mathrm{S}]] \rightarrow [(\mathrm{R},\mathrm{R}')|\mathrm{S}]}$$

The equality of $V = R'$ means that every occurrence of V in R' is replaced by R'.

Comparison

The following observation will ease the comparison of the rules for the CAM instructions and those for our generated abstract machine instructions. 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 top most element of the stack[2]. Since they have similar effects we get the following correspondence of CAM instructions and instructions of the generated abstract machine:

Generated Instruction	CAM Instruction
$\overline{\mathbf{car}}$	**car**
$\overline{\mathbf{cdr}}$	**cdr**
$\overline{\mathbf{lambda}}$	**cur**
$\overline{\mathbf{equal}}$	**push***
$\overline{\mathbf{conv\_11}}$	**swap**
$\overline{\mathbf{conv\_12}}$	**cons***
$\overline{\mathbf{run}}$	**app**
$\overline{\mathbf{conv\_14}}$	**cons***
$\overline{\mathbf{letrec}}$	**push***
$\overline{\mathbf{newind}}$;$\overline{\mathbf{conv\_15}}$	**quote**(R')**;cons;push**
$\overline{\mathbf{conv\_16}}$	**swap;rplac**

* There is not always a one-to-one correspondence of CAM instructions and generated instructions. For example, $\overline{\mathbf{equal}}$ pushes only the environment on top of the stack, whereas $\overline{\mathbf{letrec}}$ pushes both the environment and the redirections on top of the stack.

The major deviation of the generated instructions from the CAM instructions is the additional handling of redirections. In the above cited specifications of the CAM these are hidden in the meta-language, but when it comes to implementing the CAM in a language like C one has to deal with this problem.

[2] A minor notational observation is, that in the generated abstract machine we use $[\ldots|\ldots]$ to construct environments instead of $(\ldots,\ldots)$.

Chapter 7

Generation of a Compiler and Abstract Machine for Action Notation

We apply the system to a specification of Actress Action Notation (see Appendix B). As an example we trace the transformations of rules for an action and an action combinator. The resulting compiler and abstract machine can be used as a basis for a compiler generator based on Action Semantics.

7.1 Action Semantics

Action semantics [Mos92] has been developed to allow for useful semantics descriptions of realistic programming languages. The language used to write such semantics descriptions is called **action notation**.

The semantic entities of action semantics are **actions**, **data** and **yielders**. Actions are computational entities, they reflect the step-wise execution of programs. Data are mathematical entities like numbers, truth-values, lists and sets. Finally yielders represent unevaluated data. If the action containing a yielder is performed, the yielder evaluates to a concrete datum. Actions can become data by encapsulating them in **abstractions**, which can be **enacted** into actions again. The performance of an action may **complete** (i.e., normal termination), **escape** (i.e., exceptional termination which may be trapped), **fail** (i.e., abandoning the performance of an action which can lead to the performance of an alternative action) or **diverge**

(i.e., nontermination). Actions process different kinds of information and can be classified according to which **facet** they belong: **basic** (control-flow, no data are changed), **functional** (transient information, i.e., intermediate results), **declarative** (scoped information, i.e., bindings), **imperative** (stable information, i.e., the store), **communicative** (permanent information, i.e., messages send between actions), **directive** (finite representation of self-referential bindings). Actions which process information in more than one facet are called **hybrid**. An action may **commit** and discard alternatives, e.g., in an action A_1 or A_2 representing the nondeterministic choice between two sub-actions, A_1 may commit and thus A_2 is discarded. Compound actions can be build from primitive actions using a special kind of actions called **action combinators**.
Since action semantics provides so many actions and yielders we refrain from giving an exhaustive listing but instead look at some examples.
Similar to denotational semantics the action semantics of a programming language is given by semantic equations[1]:

- execute⟦ X ":=" E ⟧ = |evaluate E
  then
  |store the given value in the cell bound to X

evaluate is a semantic function defined by semantics equations similar to the way the semantic function execute is defined here. For a concrete value of E the function evaluate yields a compound action. The action combinator A_1 then A_2 propagates the transients given to the whole action to A_1, the transients given by A_1 are propagated to A_2, and only the transients given by A_2 are given by the whole action. Thus then represents the left-to-right sequencing in the functional facet. The primitive, imperative action store Y_1 in Y_2 stores the datum produced by the yielder Y_1 in the store cell (a special kind of data) produced by the yielder Y_2. Note, that items of data are a special case of yielders, and always yield themselves when evaluated. In the above example the variable name associated with X in a concrete program would be such a special yielder.

- execute⟦ "while" E "do" C "od" ⟧ = unfolding
  ||evaluate E
  |then
  |||execute C then unfold
  ||else
  |||complete

[1]Instead of using parentheses to indicate precedence of actions, in action semantics we use the convention that vertical lines group actions and their arguments.

The action combinator unfolding A performs the action A, but whenever it reaches the dummy action unfold it performs A instead. The action complete simply completes and is thus a neutral action with respect to some action combinators. The action combinator A_1 else A_2 is actually syntactic sugar for a compound action:

|| check the given truth-value and then A_1
| or
|| check no the given truth-value and then A_2

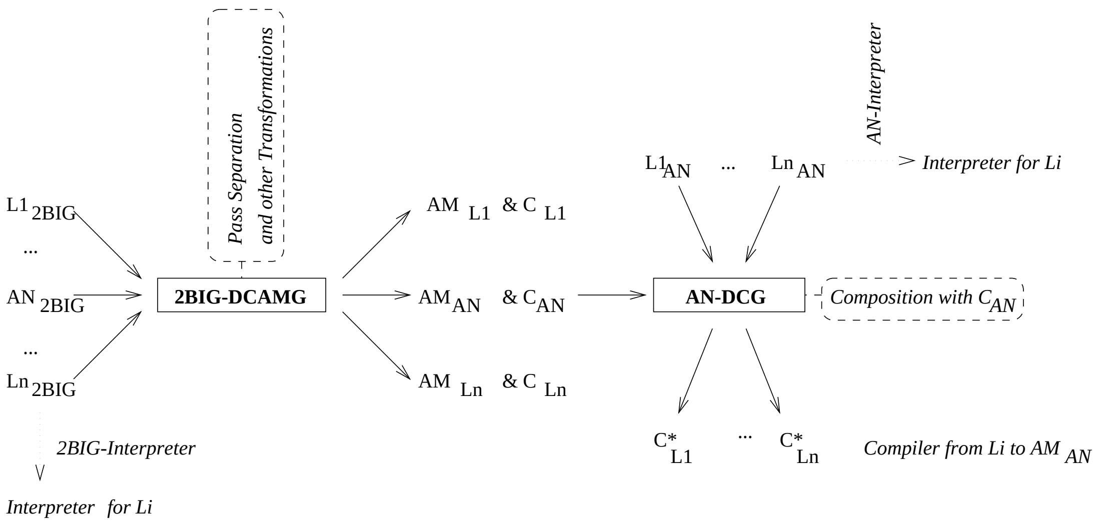
The action check Y completes if the yielder Y evaluates to true and fails if it evaluates to false. The yielder not Y evaluates to true (false) if Y evaluates to false (true). The yielder the given D evaluates to a transient datum of sort D given by a preceding action. There can be more than one transient datum, which is taken care of by a labeling mechanism. The action which gives a datum can label it, e.g., give Y label $\#n$ and later a yielder can access it, e.g., the given D label $\#n$. Now give Y is short for give Y label $\#0$ and the given D or just the D is short for the given D label $\#0$.

7.2 Transforming a 2BIG specification of Action Notation

In his PhD thesis [Mou93] deMoura gives a natural semantics specification of a subset of action notation used in the compiler generator Actress [MW94]. In this specification the order of rules is important. We converted these rules into 2BIG rules adding additional preconditions, when necessary, to make the rules determinate. Then we used our system to generate a compiler and abstract machine represented as term rewriting systems.

In Section 7.5.1 we demonstrate the generation process by transforming the 2BIG rules for the OR action combinator. In the transformation of the 2BIG rules of the GIVE action discussed in Section 7.5.2 we also deal with side conditions.

Our specification consists of 100 2BIG rules defining the semantics of 39 action notation constructs including the control, functional, declarative and imperative facets but, as in other Action Semantics directed compiler generators, neither the communicative facet, nondeterminism nor the interleaving of actions. After transformation of side conditions we got 135 rules. Factorization resulted in 191 rules. After sequentialization we got 276 rules. Finally pass separation yielded 216 compiler rules and 276 abstract machine rules. We tested this compiler and abstract machine by translating Mini-Δ programs (e.g., Fibonacci numbers) based on an action semantics specification (see Appendix A) of the language Mini-Δ [MW94] into action terms. Then we compiled these action terms using the generated compiler into an abstract machine program and executed the latter by the above abstract machine rules. In other words we use a 2BIG semantics-based compiler generator to generate a compiler and



Li_{2BIG}	2BIG specification of language Li
AN_{2BIG}	2BIG specification of action notation
$2BIG - DCAMG$	2BIG directed generator of compilers and abstract machines
AM_{Li}	abstract machine for language Li
AM_{AN}	abstract machine for action notation
C_{Li}	compiler from language Li into AM_{Li}
C_{Li}	compiler from action notation into AM_{Li}
Li_{AN}	action semantics specification of language Li
C^*_{Li}	compiler from language Li into AM_{AN}
$AN - DCG$	action semantics directed compiler generator

Figure 7.1: Action-Semantics Directed Compiler Generation

abstract machine for action notation. The generated compiler is then inserted as the back end into a compiler generator based on action semantics (see Figure 7.1). The front end of this compiler generator was previously developed and used with a positive supercompiler for Prolog[2] as its back end [Diea].

7.3 Prototyping Tools

In the picture 7.1 we show how the different generators and interpreters can be used for both rapid prototyping of language specifications and generation of compilers and abstract machines. First we can use a 2BIG-interpreter to test a 2BIG-specification Li_{2BIG} of programming languages Li. Then we can generate an abstract machine AM_{Li} for the language Li and a compiler C_{Li} from Li to AM_{Li} using our 2BIG-semantics directed compiler and abstract machine generator (2BIG-DCAMG). The generator's central transformation is pass separation of term rewriting rules. In addition it applies many pre- and post-processing transformations including several optimizations.
Based on a 2BIG-specification AN_{2BIG} of a certain language, namely action notation, we generate a compiler and abstract machine for action notation. Now an action semantics specification Li_{AN} of a programming language Li can be tested both by using an action notation interpreter or by composing the action notation specification,i.e., semantics equations mapping Li programs to action notation terms, with the compiler C_{AN}. This composition results in an action semantics-directed compiler generator (AN-DCG).
Our prototyping environment includes several tools written in Prolog:

- a 2BIG interpreter
- an action notation interpreter (actually we have a handwritten interpreter, but we can also use the 2BIG interpreter to execute action terms using the 2BIG specification of action notation)
- an interpreter for compiler and abstract machine rules
- a compiler of source language programs to C using the compiler and abstract machine rules
- a compiler of compiler rules and abstract machine rules to SML

[2] Positive supercompilation [Tur86, SGJ94] is a program specialization technique developed in the functional community. Its adaption to Prolog is not much different from partial evaluation of Prolog [GS94].

7.4 An Abstract Machine Language Language

Since Action Semantics is a formal language to define programming languages, we expect, that the abstract machine language AM_{AN} generated for Action Semantics is suitable to define abstract machine languages. Rather than just composing the AM_{AN} and the semantics equation which gives us AN-DCG, we could try a method similar to the combinator based approach (see section 3.11.2). Given an Action Semantics specification of a programming language L:

1. Translate the right hand sides of the semantics equation using the generated compiler into AM_{AN} (this results in an AN-DCG).
2. Look for recurring patterns in the translated right hand side.
3. Define new instructions based on these patterns. These new instructions form an abstract machine specific for L.
4. Fold the patterns in the semantics equations by the new instructions. The resulting equations constitute a compiler into the abstract machine for L.

7.5 Action Semantics-Directed Compiler Generation

Now we will show how our action semantics-based compiler generator works by means of a simple example. The semantics of the language Mini-Δ is given by equations like the following one:

- execute⟦ X ":=" E ⟧ =
  |evaluate E
  then store the value in the cell bound to X

These semantic equations define a translation function from source language programs to action terms. Using this action semantics specification of Mini-Δ the following program

```
let const i=1;
    var   x:integer;
in x:=2+i end
```

is translated into the following action term

- ```
  |furthermore
  |||give num(1) then bind i to the given value
  ||before
  |||allocate a cell of type integer then bind x to the cell
  hence
  ||||give num(2) then give the value label#1
  |||and
  ||||||give the value stored in the cell bound to i
  |||||or
  ||||||give the value bound to i
  ||||then
  |||||give the value label #2
  ||then
  |||give add(the value #1,the value #2)
  |then
  ||store the given value in the cell bound to x
  ```

In our system we use prefix notation instead of the mixfix notation usually used for action terms. Thus A_1 **then** A_2 becomes **then**(A_1, A_2). The above action term in prefix notation is:

```
hence(
  furthermore(
      before(then(give(num(1),0),bind(i,the(value,0))),
             then(allocate(cell(integer)),bind(x,the(cell,0))))),
   then(
     then(
       and(then(give(num(2),0),give(the(value,0),1)),
           then(or(give(stored(value,bound(cell,i)),0),
                   give(bound(value,i),0)),
                give(the(value,0),2))),
       give(add(the(value,1),the(value,2)),0)),
     store(the(value,0),bound(cell,x))))
```

Now this action term is converted into a very long abstract machine program by the generated compiler. One reason for the length of the abstract machine program is that recurring subprograms are not shared.
```

$$
\begin{array}{l}
\overline{hence}(\\
\quad \overline{furthermore}(\\
\quad\quad \overline{before}(\\
\quad\quad\quad \overline{then}(\\
\quad\quad\quad\quad (\overline{give}(\overline{num}(1), 0); \\
\quad\quad\quad\quad \overline{num}(1); \\
\quad\quad\quad\quad \overline{conv}_2; \\
\quad\quad\quad\quad \overline{test}_1; \\
\quad\quad\quad\quad \overline{conv}_3; \\
\quad\quad\quad\quad \overline{fact}_{give}(0)), \\
\quad\quad\quad\quad (\overline{bind}(i, \overline{the}(value, 0)); \\
\quad\quad\quad\quad \overline{the}(value, 0); \\
\quad\quad\quad\quad \overline{conv}_1 4; \\
\quad\quad\quad\quad \overline{test}_5; \\
\quad\quad\quad\quad \overline{conv}_1 5; \\
\quad\quad\quad\quad \overline{fact}_{bind}(i))); \\
\quad\quad\quad\quad \overline{give}(\overline{num}(1), 0); \\
\quad\quad\quad\quad \ldots
\end{array}
$$

The execution of the above program by the abstract machine in the empty environment yields the expected result: a memory cell is allocated for the variable `x` and the value 3 is stored in it.

In the above example the action term could be simplified before translating it into the abstract machine language. As an example give num(2) then give the value label#1 can be simplified to give num(2) label#1. Analyses and simplifications of action terms have been investigated in de Moura's PhD thesis [Mou93]. It would be interesting to use the simplified action terms produced by his Actress system and translate those into the generated abstract machine language.

In the rest of this chapter we will demonstrate step by step how our system generated the abstract machine instructions for the OR action combinator and the GIVE action.

7.5.1 Transforming the OR Action Combinator

In the 2BIG specification the following rules define the action combinator *or*. In the rules, states are composed of the transients T, the bindings B and a single-threaded store S. Furthermore there is the outcome status O, which can be *failed* or *completed*.

$$\frac{A_1 \triangleright [T,B,S] \rightarrow [completed,T_1,B_1,S_1]}{or(A_1,A_2) \triangleright [T,B,S] \rightarrow [completed,T_1,B_1,S_1]}$$

$$\frac{A_1 \triangleright [T,B,S] \rightarrow [failed,[],[],S_1] \qquad A_2 \triangleright [T,B,S_1] \rightarrow [O_2,T_2,B_2,S_2]}{or(A_1,A_2) \triangleright [T,B,S] \rightarrow [O_2,T_2,B_2,S_2]}$$

There are no side conditions, so the above rules are next factorized:

$$\frac{A_1 \triangleright [T,B,S] \rightarrow [O_1,T_1,B_1,S_1] \qquad fact_{or}(A_2) \triangleright [[B,T],[O_1,T_1,B_1,S_1]] \rightarrow E}{or(A_1,A_2) \triangleright [T,B,S] \rightarrow E}$$

$$fact_{or}(A) \triangleright [[B,T],[completed,T_1,B_1,S_1]] \rightarrow [completed,T_1,B_1,S_1]$$

$$\frac{A \triangleright [T,B,S] \rightarrow [O,T_1,B_1,S_1]}{fact_{or}(A) \triangleright [[B,T],[failed,[],[],S] \rightarrow [O,T_1,B_1,S_1]}$$

Now the stack (Z) is introduced and temporary variables are allocated, e.g., in the first rule T,B are allocated on the stack, because they do not occur on the right side of the first precondition.

$$\frac{\begin{array}{l} A_1 \triangleright [[[T,B]|Z],[T,B,S]] \rightarrow [[[T,B]|Z],[O_1,T_1,B_1,S_1]] \\ fact_{or}(A_2) \triangleright [Z,[[B,T],[O_1,T_1,B_1,S_1]]] \rightarrow [Z,E] \end{array}}{\mathrm{or}(\mathrm{A}_1,\mathrm{A}_2) \triangleright [\mathrm{Z},[\mathrm{T},\mathrm{B},\mathrm{S}]] \rightarrow [\mathrm{Z},\mathrm{E}]}$$

$$fact_{or}(A) \triangleright [Z,[[B,T],[completed,T_1,B_1,S_1]]] \rightarrow [Z,[completed,T_1,B_1,S_1]]$$

$$\frac{A \triangleright [Z,[T,B,S]] \rightarrow [Z,[O,T_1,B_1,S_1]]}{fact_{or}(A) \triangleright [Z,[[B,T],[failed,[],[],S]] \rightarrow [Z,[O,T_1,B_1,S_1]]}$$

Next these rules are sequentialized:

$$\frac{\begin{array}{l} A_1 \triangleright [[[T,B]|Z],[T,B,S]] \rightarrow [[[T,B]|Z],[O_1,T_1,B_1,S_1]] \\ conv_1 \triangleright [[[T,B]|Z],[O_1,T_1,B_1,S_1]] \rightarrow [Z,[[B,T],[O_1,T_1,B_1,S_1]]] \\ fact_{or}(A_2) \triangleright [Z,[[B,T],[O_1,T_1,B_1,S_1]]] \rightarrow [Z,E] \end{array}}{\mathrm{or}(\mathrm{A}_1,\mathrm{A}_2) \triangleright [\mathrm{Z},[\mathrm{T},\mathrm{B},\mathrm{S}]] \rightarrow [\mathrm{Z},\mathrm{E}]}$$

$$fact_{or}(A) \triangleright [Z,[[B,T],[completed,T_1,B_1,S_1]]] \rightarrow [Z,[completed,T_1,B_1,S_1]]$$

$$\frac{A \triangleright [Z,[T,B,S]] \rightarrow [Z,[O,T_1,B_1,S_1]]}{fact_{or}(A) \triangleright [Z,[[B,T],[failed,[],[],S]] \rightarrow [Z,[O,T_1,B_1,S_1]]}$$

$$conv_1 \triangleright [[[T,B]|Z],[O_1,T_1,B_1,S_1]] \rightarrow [Z,[[B,T],[O_1,T_1,B_1,S_1]]]$$

Now a term rewriting system is generated:

$$\begin{array}{lcl}
\langle or(A_1,A_2);C,[Z,[T,B,S]]\rangle \;\Rightarrow\; \langle A_1;conv_1;fact_{or}(A_2);C,[[[B,T]|Z],[T,B,S]]\rangle & & \\
\langle fact_{or}(A);C,[Z,[[T,B],[completed,T_1,B_1,S_1]]]\rangle & \Rightarrow & \langle C,[Z,[completed,T_1,B_1,S_1]]\rangle \\
\langle fact_{or}(A);C,[Z,[[T,B],[failed,[],[],S]]]\rangle & \Rightarrow & \langle A;C,[Z,[T,B,S]]\rangle \\
\langle conv_1;C,[[B,T]|Z],[O,T_1,B_1,S_1]]\rangle & \Rightarrow & \langle C,[Z,[[T,B],[O,T_1,B_1,S_1]]]\rangle
\end{array}$$

Finally we apply the pass separation transformation and we get the following compiler rules:

$$\begin{array}{lcl}
or(A_1,A_2) & \Rightarrow & \overline{or}(A_1,A_2);A_1;conv_1;fact_{or}(A_2) \\
fact_{or}(A) & \Rightarrow & \overline{fact}_{or}(A) \\
conv_1 & \Rightarrow & \overline{conv}_1
\end{array}$$

And the following abstract machine rules:

$$\begin{array}{lcl}
\langle \overline{or}(A_1,A_2);C,[Z,[T,B,S]]\rangle & \Rightarrow & \langle C,[[[B,T]|Z],[T,B,S]]\rangle \\
\langle \overline{fact}_{or}(A);C,[Z,[[T,B],[completed,T_1,B_1,S_1]]]\rangle & \Rightarrow & \langle C,[Z,[completed,T_1,B_1,S_1]]\rangle \\
\langle \overline{fact}_{or}(A);C,[Z,[[T,B],[failed,[],[],S]]]\rangle & \Rightarrow & \langle A;C,[Z,[T,B,S]]\rangle \\
\langle \overline{conv}_1;C,[[B,T]|Z],[O,T_1,B_1,S_1]]\rangle & \Rightarrow & \langle C,[Z,[[T,B],[O,T_1,B_1,S_1]]]\rangle
\end{array}$$

7.5.2 Transforming the GIVE Action

In the 2BIG specification the following rules define the action *give* which evaluates the yielder Y and returns the resulting value D as a transient:

$$\frac{Y \triangleright [T,B,S] \rightarrow datum(D) \qquad D \neq nothing}{give(Y,N) \triangleright [T,B,S] \rightarrow [completed,[N \mapsto datum(D)],[],S]}$$

$$\frac{Y \triangleright [T,B,S] \rightarrow datum(D) \qquad not(D \neq nothing)}{give(Y,N) \triangleright [T,B,S] \rightarrow [failed,[],[],S]}$$

There are two side conditions in the above rules, one is the negation of the other. Transforming the side conditions yields:

$$\frac{Y \triangleright [T,B,S] \rightarrow datum(D) \qquad test_1 \triangleright [D] \rightarrow true}{give(Y,N) \triangleright [T,B,S] \rightarrow [completed,[N \mapsto datum(D)],[],S]}$$

$$\frac{Y \triangleright [T,B,S] \rightarrow datum(D) \qquad test_1 \triangleright [D] \rightarrow false}{give(Y,N) \triangleright [T,B,S] \rightarrow [failed,[],[],S]}$$

$$test_1 \triangleright [D] \rightarrow D \neq nothing$$

After factorization of the above rules we have:

$$\frac{Y \triangleright [T,B,S] \rightarrow datum(D) \qquad test_1 \triangleright [D] \rightarrow R \qquad fact_{give}(N) \triangleright [[D,S],R] \rightarrow E}{give(Y,N) \triangleright [T,B,S] \rightarrow E}$$

$$fact_{give}(N) \triangleright [[D,S], true] \rightarrow [completed, [N \mapsto datum(D)], [], S]$$

$$fact_{give}(N) \triangleright [[D,S], false] \rightarrow [failed, [], [], S]$$

$$test_1 \triangleright [D] \rightarrow D \neq nothing$$

Now the stack (Z) is introduced and temporary variables are allocated:

$$\frac{\begin{array}{l} Y \triangleright [[S|Z],[T,B,S]] \rightarrow [[S|Z], datum(D)] \\ test_1 \triangleright [[[S,D]|Z],[D]] \rightarrow [[[S,D]|Z],R] \qquad fact_{give}(N) \triangleright [Z,[[D,S],R]] \rightarrow [Z,E] \end{array}}{\text{give(Y,N)}\triangleright\text{[Z,[T,B,S]]}\rightarrow\text{[Z,E]}}$$

$$fact_{give}(N) \triangleright [Z,[[D,S], true]] \rightarrow [Z,[completed, [N \mapsto datum(D)], [], S]]$$

$$fact_{give}(N) \triangleright [Z,[[D,S], false]] \rightarrow [Z,[failed, [], [], S]]$$

$$test_1 \triangleright [Z,[D]] \rightarrow [Z, D \neq nothing]$$

Next these rules can be sequentialized:

$$\frac{\begin{array}{ll} Y \triangleright [[S|Z],[T,B,S]] \rightarrow [[S|Z], datum(D)] & \\ conv_5 \triangleright [[S|Z], datum(D)] \rightarrow [[[S,D]|Z],[D]] & test_1 \triangleright [[[S,D]|Z],[D]] \rightarrow [[[S,D]|Z],R] \\ conv_6 \triangleright [[[S,D]|Z],R] \rightarrow [Z,[[D,S],R]] & fact_{give}(N) \triangleright [Z,[[D,S],R]] \rightarrow [Z,E] \end{array}}{\text{give(Y,N)}\triangleright\text{[Z,[T,B,S]]}\rightarrow\text{[Z,E]}}$$

$$fact_{give}(N) \triangleright [Z,[[D,S], true]] \rightarrow [Z,[completed, [N \mapsto datum(D)], [], S]]$$

$$fact_{give}(N) \triangleright [Z,[[D,S], false]] \rightarrow [Z,[failed, [], [], S]]$$

$test_1 \triangleright [Z, [D]] \rightarrow [Z, D \neq nothing]$

$conv_2 \triangleright [[S|Z], datum(D)] \rightarrow [[[S, D]|Z], [D]]$

$conv_3 \triangleright [[[S, D]|Z], R] \rightarrow [Z, [[D, S], R]]$

Now a term rewriting system is generated:

$$\begin{array}{lll}
\multicolumn{3}{l}{\langle give(Y, N); C, [Z, [T, B, S]]\rangle} \\
\multicolumn{3}{l}{\qquad\quad \Longrightarrow \ \langle Y; conv_2; test_1; conv_3; fact_{give}(N); C, [[[S]|Z], [T, B, S]]\rangle} \\
\langle fact_{give}(N); C, [Z, [[D, S], true]] & \Longrightarrow & \langle C, [Z, [completed, [N \mapsto datum(D)], [], S]]\rangle \\
\langle fact_{give}(N); C, [Z, [[D, S], false]] & \Longrightarrow & \langle C, [Z, [failed, [], [], S]]\rangle \\
\langle test_1; C, [Z, [D]] & \Longrightarrow & \langle C, [Z, D \neq nothing]\rangle \\
\langle conv_2; C, [[[S]|Z], datum(D)]\rangle & \Longrightarrow & \langle C, [[[S, D]|Z], [D]]\rangle \\
\langle conv_3; C, [[[S, D]|Z], R]\rangle & \Longrightarrow & \langle C, [Z, [[D, S], R]]\rangle
\end{array}$$

Finally we apply the pass separation transformation and we get the following compiler rules:

$$\begin{array}{lll}
give(Y, N) & \Longrightarrow & \overline{give}(Y, N); Y; conv_2; test_1; conv_3; fact_{give}(N) \\
fact_{give}(N) & \Longrightarrow & \overline{fact}_{give}(N) \\
test_1 & \Longrightarrow & \overline{test}_1 \\
conv_2 & \Longrightarrow & \overline{conv}_2 \\
conv_3 & \Longrightarrow & \overline{conv}_3
\end{array}$$

And the following abstract machine rules:

$$\begin{array}{lll}
\langle \overline{give}(Y, N); C, [Z, [T, B, S]]\rangle & \Longrightarrow & \langle C, [[[S]|Z], [T, B, S]]\rangle \\
\langle \overline{fact}_{give}(N); C, [Z, [[D, S], true]]\rangle & \Longrightarrow & \langle C, [Z, [completed, [N \mapsto datum(D)], [], S]]\rangle \\
\langle \overline{fact}_{give}(N); C, [Z, [[D, S], false]]\rangle & \Longrightarrow & \langle C, [Z, [failed, [], [], S]]\rangle \\
\langle \overline{test}_1; C, [Z, [D]]\rangle & \Longrightarrow & \langle C, [Z, D \neq nothing]\rangle \\
\langle \overline{conv}_2; C, [[[S]|Z], datum(D)]\rangle & \Longrightarrow & \langle C, [[[S, D]|Z], [D]]\rangle \\
\langle \overline{conv}_3; C, [[[S, D]|Z], R]\rangle & \Longrightarrow & \langle C, [Z, [[D, S], R]]\rangle
\end{array}$$

7.6 Experimental Results for Optimizations

For the action notation specification, the optimizations of the generated term rewriting systems lead to a significant reduction of the number of rules both of the compiler and the abstract machine. First, by self-application, the number of compiler rules was reduced from 216 to 43. Second, using the other optimizations we got 181 instead of 276 abstract machine rules.

$$give(Y,N) \implies \overline{give}(Y,N); Y; conv_2; test_1; conv_3; fact_{give}(N)$$

$$give(Y,N) \implies \overline{give}; Y; \overline{comb}; \overline{fact}_{disp}(factor_{give}, N)$$

Comparing the original and the optimized compiler rule for the GIVE action we find that the following optimizations have been applied:

- The arguments to the abstract machine instruction $\overline{give}$ have been removed.
- There are no more compiler rules for $conv_2$, $test_1$, etc.
- The sequence of instructions $\overline{conv}_2; \overline{test}_1; \overline{conv}_3$ has been combined into the instruction $\overline{comb}$.
- Some abstract machine rules of $factor_{give}$ have been conflicting with rules of other instructions and thus these term rewriting rules have been factorized. This lead to the introduction of the new instruction $\overline{fact}_{disp}$.

7.7 Designing Semantics Formalisms

In this thesis our prototyping tools have been used to implement a considerable subset of Action Semantics. Instead one could also try to implement subsets of Action Semantics restricted to a few facets. As an example, to specify functional languages we could implement a version of Action Semantics without the imperative facet. As a consequence the generated abstract machine would not have a store as a compenent of its state. Another approach would be to implement an annotated version of Action Semantics and a preprocessing phase, e.g., a binding-time analysis, which translates action terms into annotated terms. Finally, rather than just experimenting with existing semantics formalism, our system can also be used to design and implement new semantics formalisms.

Chapter 8

Performance Evaluation

The main focus of our work has been feasability 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.

8.1 Introduction

For the benchmarks we used a 60Mhz Pentium PC with 8MB RAM running the Linux Operating System. On this machine GCC (version 2.4.5), Standard ML of New Jersey (version 0.93) and Sicstus Prolog (version 2.1) have been available. We did similar benchmarks on a Solbourne (Series 5) wich was about 1.4 times slower and a Sun Sparc 20 which was about 1.2 times faster.

The following benchmarks were run:

- *loop*: a program which counts form 1 to n.
- *fib*: a program which computes the Fibonacci number of n.
- *primes*: a program which computes the first n prime numbers using the algorithm known as "Sieve of Eratosthenes".

We did benchmarks with three different languages. Only for SIMP we tested all three programs:

- SIMP: *loop*, an iterative *fib* program and *primes*
- Mini-ML: *loop* and a recursive *fib* program
- Action Notation: action terms produced by expanding Mini-Δ programs of *loop* and a recursive *fib*

For these programs, times of the following ways of execution are given:

- Interpreting the 2BIG rules in Prolog
- Executing an abstract machine program on the generated abstract machine in SML
- Executing an abstract machine program on the generated abstract machine in C

To compare our results to production quality compilers we also provide the execution times of an SML program for Fibonacci numbers and a C program for prime numbers.

8.2 Performance Results

Execution Times for the *loop* Program									
	SIMP			Mini-ML			Mini-Δ		
n	2BIG in Prolog	AM in C	AM in SML	2BIG in Prolog	AM in C	AM in SML	2BIG in Prolog	AM in C	AM in SML
10	1.25	< 0.05	0.01	4.50	<0.10	0.01	22.80	0.20	2.40
50	5.80	<0.05	0.11	21.50	<0.10	0.17	101.60	1.30	42.40
100	11.50	<0.05	0.40	42.70	<0.10	0.56	201.90	2.70	166.75
1000		0.40	31.1		1.30	46.76		29.80	
10000		5.00							

Execution Times for the *fib* Program									
	SIMP			Mini-ML			Mini-Δ		
n	2BIG in Prolog	AM in C	AM in SML	2BIG in Prolog	AM in C	AM in SML	2BIG in Prolog	AM in C	AM in SML
3	1.43	<0.01	<0.01	4.40	<0.05	<0.05	121.60	0.10	1.44
5	2.25	<0.01	0.01	13.50	<0.05	0.04	362.1	0.40	9.73
6	2.70	<0.01	0.01	22.50	<0.05	0.06	633.3	0.70	24.53
7	3.10	<0.01	0.02	37.4	<0.05	0.11		1.30	63.01
8	3.50	<0.01	0.02	61.4	0.08	0.24		2.40	168.68
9	3.95	0.02	0.02		0.10	0.55		4.40	449.36
12		0.02	0.03		0.40	5.90			
15		0.02	0.04		2.00				
20		0.03	0.05		24.80				
40		0.04	0.12						
100		0.09							

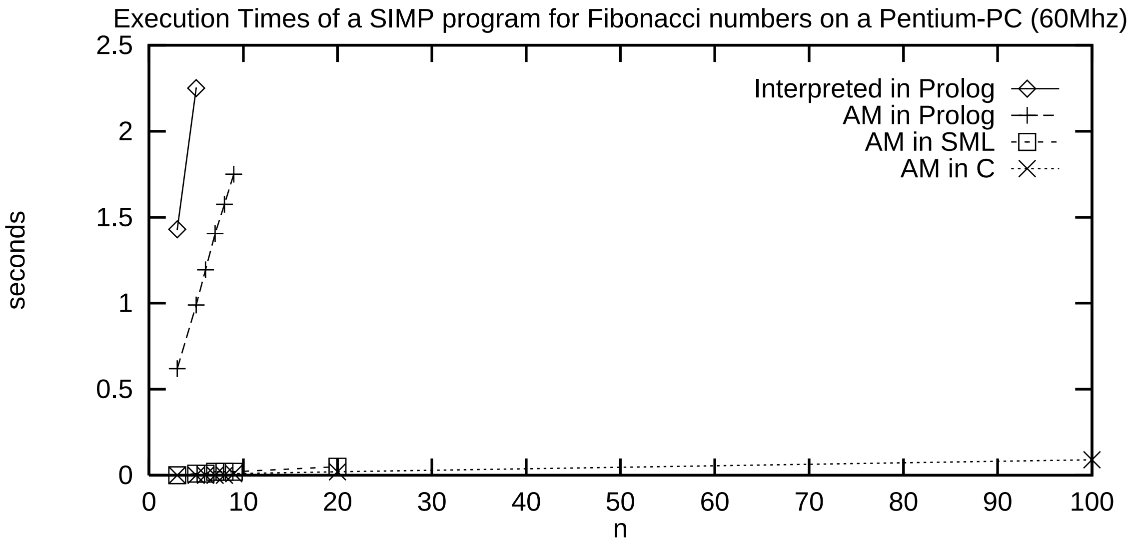


Figure 8.1: SIMP: Fibonacci numbers

SIMP

As shown in Figure 8.1 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 an equivalent C program.

Execution Times for the *prime* Program			
	SIMP		C
n	AM in C	AM in SML	
30	< 0.01	0.34	
100	0.20	4.18	
500	1.70		< 0.01
1000	4.00		0.03

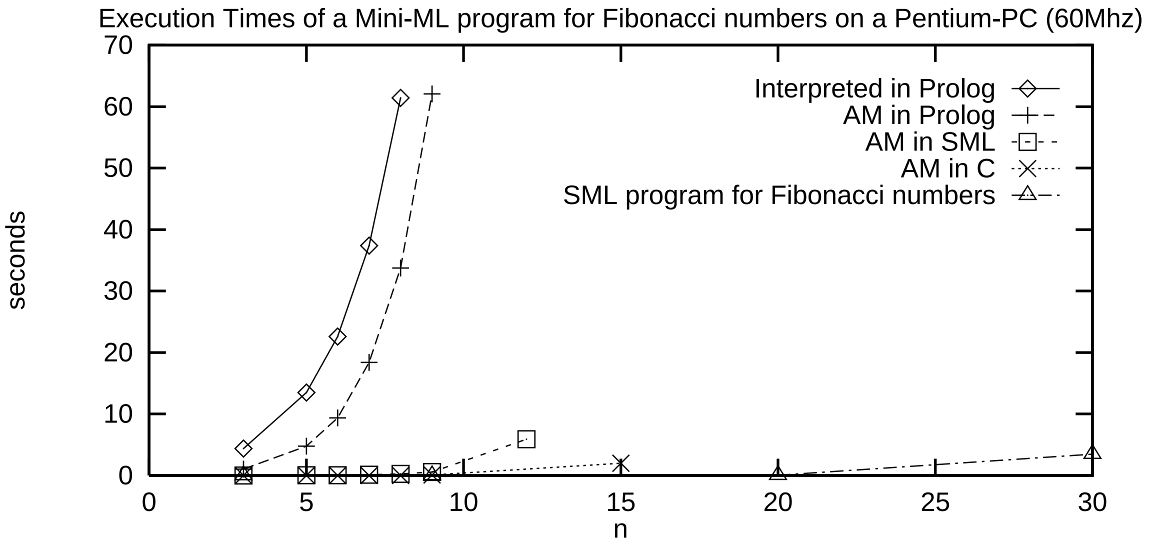


Figure 8.2: Mini-ML: Fibonacci numbers

Mini-ML

In Figure 8.2 we see that the generated abstract machine 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 SML program computing Fibonacci numbers compiled with the SML of New Jersey Compiler.

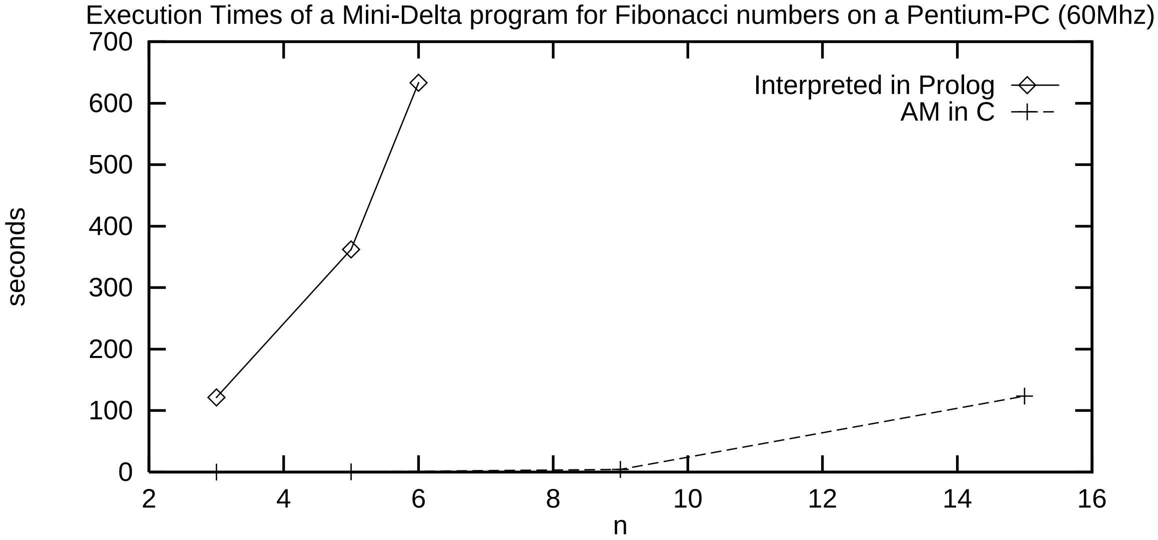


Figure 8.3: Mini-Δ: Fibonacci numbers

Action Notation of Mini-Δ

Mini-Δ 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 these performance tests a benchmark program in the language Mini-Δ was first translated into an action term using the Action Semantics specification of Mini-Δ in Appendix A. 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. As shown in Figure 8.3, the abstract machine in C is at least 2 orders of magnitude faster.

Chapter 9

Correctness of Transformations

"Every program has at least one bug and can be shortened by at least one instruction – from which, by induction, one can deduce that every program can be reduced to one instruction which doesn't work. "

/usr/games/fortune

"Contrariwise," continued Tweedledee, "if it was so, it might be, and if it were so, it would be; but as it isn't, it ain't. That's logic!"

– Lewis Carroll, "Through the Looking Glass"

First we discuss the importance of doing both correctness proofs and experimental evaluation. To prove the correctness of our transformations we refer to the formal definitions of rule induction in Section 4.3 and term rewriting rules in Section 5.1.1. Furthermore we need the properties of 2BIG rules and term rewriting rules as defined in Section 4.5 and 5.1.2. Before we actually prove the correctness of the transformations of the core system, we explain our basic proof technique.

9.1 Correctness and Experimental Evaluation

We feel that a system of considerable complexity needs both experimental evaluation and correctness proofs. Given the complexity of the system neither experimental evaluation nor the correctness proofs are absolutely reliable. Combining these conceptually different approaches helps to decrease the number of errors in the system.
That programs which have been proved correct can go wrong, is best expressed in Knuth's famous quote:

> "Beware of bugs in the above code; I have only proved it correct, not tried it. "
>
> Donald E. Knuth

Such bugs might be syntax errors, machine dependent errors or ignored side-effects.
As programs and especially large programs tend to be error-prone so are proofs. And as syntactic and semantic program analysis helps to find bugs, so might proof checkers assist to make correctness proofs correct.

> "The correctness statement, including various lemmas but without proofs, takes 28 pages. Putting further sophistication into the compiler will add significantly to these page counts. We feel that the size alone of the specifications calls for automatic proof checking. "
>
> Jens Palsberg [Pal92b, page 11]

On the other hand experimental evaluation seems to be a much simpler task[1] and thus should

[1]Though who ever thought about finding "covering" test examples certainly will admit that this is not at all trivial.

be a minimum requirement. Curiously enough, as a recent study revealed, even experimental evaluation is not standard in computer science.

> "Of the papers in the random sample that would require experimental validation, 40% have none at all. In journals related to software engineering this fraction is over 50%. "
>
> Luckowicz et al. [LHPT94]

9.2 Proof Technique

The transformations we consider here 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 $T_\Sigma(X)$.
We will prove claims of the form:

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

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

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

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

Our correctness proofs can be based on $\overline{\phi}$-proofs using Theorem 4.3.7 or on $\overline{\phi}$-trees using Theorem 4.3.9. Since proof trees provide more structure we use them here.

Proof:

Completeness: $x \in \mathcal{I}(\overline{\phi})$

$\overset{4.3.9}{\Longrightarrow}$ there exists a $\overline{\phi}$-tree $\dfrac{PT_{\overline{\phi}}(b_1)\ \ldots\ PT_{\overline{\phi}}(b_n)}{x}$ for x

$\Rightarrow$ $\boxed{\text{Here the definitions of the transformation } T \text{ and the relation } R \text{ are used to construct a } \overline{\phi'}\text{-tree. We use induction on the depth of the } \overline{\phi}\text{-tree}}$

$\Rightarrow \dfrac{PT_{\overline{\phi'}}(b'_1)\ \ldots\ PT_{\overline{\phi'}}(b'_m)}{y}$ is a $\overline{\phi'}$-tree for y

$\stackrel{4.3.9}{\Longrightarrow} y \in \mathcal{I}(\overline{\phi'})$

Soundness: $y \in \mathcal{I}(\overline{\phi'})$

$\stackrel{4.3.9}{\Longrightarrow}$ there exists a $\overline{\phi'}$-tree $\dfrac{PT_{\overline{\phi'}}(b'_1)\ \ldots\ PT_{\overline{\phi'}}(b'_m)}{y}$ for y

$\Rightarrow$ Here the definitions of the transformation T and the relation R are used to construct a $\overline{\phi}$-tree. We use induction on the depth of the $\overline{\phi'}$-tree

$\Rightarrow \dfrac{PT_{\overline{\phi}}(b_1)\ \ldots\ PT_{\overline{\phi}}(b_n)}{x}$ is a $\overline{\phi}$-tree for x

$\stackrel{4.3.9}{\Longrightarrow} x \in \mathcal{I}(\overline{\phi})$

A remark on the induction principle is in order. The induction principle is $((\forall n < d : P(n)) \Rightarrow P(d)) \Rightarrow \forall d : P(d)$. Using this version of the induction principle, we do not have to prove the base case $d = 1$ separately.

9.3 Correctness Proofs

9.3.1 Correctness of Stack Introduction and Allocation of Temporary Variables

Let T_1 be the stack introduction transformation and T_2 be the allocation of temporary variables. We now prove the correctness of the composed transformation $T_A = T_2 \circ T_1$, i.e., $T_A(\phi) = T_2(T_1(\phi))$.

Theorem 9.3.1 *Let ϕ be a set of* 2BIG *rules and $\phi' = T_A(\phi)$.*
Then $c \rhd e \rightarrow e' \in \mathcal{I}(\overline{\phi}) \Leftrightarrow c \rhd [s, e] \rightarrow [s, e'] \in \mathcal{I}(\overline{\phi'})$ for some $s \in T_\Sigma$.

We prove two stronger lemmata. First, given a $\overline{\phi}$-tree we can construct $\overline{\phi'}$-trees for all possible values of the stack and second, given a $\overline{\phi'}$-tree we can reconstruct the original $\overline{\phi}$-tree by simply ignoring the stack.

Lemma 9.3.2 *If* $\frac{PT_{\overline{\phi}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1) \ \ldots \ PT_{\overline{\phi}}(\overline{c}_n \triangleright \overline{e}_n \rightarrow \overline{e}'_n)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$ *is a* $\overline{\phi}$*-tree of depth* d *then for every* $\overline{s} \in T_\Sigma$ *there is a* $\overline{\phi'}$*-tree* $\frac{PT_{\overline{\phi'}}(\overline{c}_1 \triangleright [\overline{s^*_1}, \overline{e}_1] \rightarrow [\overline{s^*_1}, \overline{e}'_1]) \ \ldots \ PT_{\overline{\phi'}}(\overline{c}_n \triangleright [\overline{s^*_n}, \overline{e}_n] \rightarrow [\overline{s^*_n}, \overline{e}'_n])}{\overline{c} \triangleright [\overline{s}, \overline{e}] \rightarrow [\overline{s}, \overline{e}']}$ *of depth* d *where* $\overline{s^*_i} = \overline{s}$ *or* $\overline{s^*_i} = [\overline{v_i} | \overline{s}]$ *for some* $\overline{v}_i \in T_\Sigma$.

The terms $\overline{s^*_i}$ depend on whether the original rule contained temporary variables or not.

Proof. The proof is by induction on the depth d of the $\overline{\phi}$-tree.

Induction Hypothesis: Lemma holds for all $n < d$.

Induction Step: Given the $\overline{\phi}$-tree $\frac{PT_{\overline{\phi}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1) \ \ldots \ PT_{\overline{\phi}}(\overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$

$\overset{4.3.8}{\Longrightarrow} \exists \overline{r} = (\overline{B}, \overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \in \overline{\phi}$ where $\overline{B} = \{\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1, \ldots, \overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m)\}$

$\overset{4.3.13}{\Longrightarrow} \exists \theta. \exists r = (B, c \triangleright e \rightarrow e') \in \phi : \eta_\sigma(\theta(r)) = \overline{r}$

$\Rightarrow T_A(r) = (B', c \triangleright [X, e] \rightarrow [X, e']) \in \phi'$ where X is a new variable and $s^* = X$ or $s^* = [v|X]$ for some term v and $B' = \{c_1 \triangleright [s^*, e_1] \rightarrow [s^*, e'_1], \ldots, c_m \triangleright [s^*, e_m] \rightarrow [s^*, e'_m]\}$

$\overset{4.3.13}{\Longrightarrow}$ Let τ be a ground substitution for X, $\tau(X) = \overline{s}$

then $\eta_\sigma(\tau(\theta(T_A(r)))) \in \overline{\phi'}$

$\Rightarrow$ The depth of every $\overline{\phi}$-tree $PT_{\overline{\phi}}(\overline{c}_k \triangleright \overline{e}_k \rightarrow \overline{e}'_k)$ is smaller or equal to n. By the induction hypothesis there exist $\overline{\phi'}$-trees $PT_{\overline{\phi'}}(\overline{c}_k \triangleright [\overline{s^*}, \overline{e}_k] \rightarrow [\overline{s^*}, \overline{e}'_k])$ with the same depth.

$\overset{4.3.8}{\Longrightarrow} \frac{PT_{\overline{\phi'}}(\overline{c}_1 \triangleright [\overline{s^*}, \overline{e}_1] \rightarrow [\overline{s^*}, \overline{e}'_1]) \ \ldots \ PT_{\overline{\phi'}}(\overline{c}_m \triangleright [\overline{s^*}, \overline{e}_m] \rightarrow [\overline{s^*}, \overline{e}'_m])}{\eta_\sigma(\tau(\theta(c \triangleright [s, e] \rightarrow [s, e'])))}$

$= \frac{PT_{\overline{\phi'}}(\overline{c}_1 \triangleright [\overline{s^*}, \overline{e}_1] \rightarrow [\overline{s^*}, \overline{e}'_1]) \ \ldots \ PT_{\overline{\phi'}}(\overline{c}_m \triangleright [\overline{s^*}, \overline{e}_m] \rightarrow [\overline{s^*}, \overline{e}'_m])}{\overline{c} \triangleright [\overline{s}, \overline{e}] \rightarrow [\overline{s}, \overline{e}']}$ is a $\overline{\phi'}$-tree of depth d

□

Lemma 9.3.3 *Let* $\overline{s} \in T_\Sigma$ *be a ground term. If* $\frac{PT_{\overline{\phi'}}(\overline{c}_1 \triangleright [\overline{s^*_1}, \overline{e}_1] \rightarrow [\overline{s^*_1}, \overline{e}'_1]) \ \ldots \ PT_{\overline{\phi'}}(\overline{c}_n \triangleright [\overline{s^*_n}, \overline{e}_n] \rightarrow [\overline{s^*_n}, \overline{e}'_n])}{\overline{c} \triangleright [\overline{s}, \overline{e}] \rightarrow [\overline{s}, \overline{e}']}$ *is a* $\overline{\phi'}$*-tree of depth* d *where* $\overline{s^*_i} = \overline{s}$ *or* $\overline{s^*_i} = [\overline{v_i} | \overline{s}]$ *for some* $\overline{v_i} \in T_\Sigma$ *then there is a* $\overline{\phi}$*-tree* $\frac{PT_{\overline{\phi'}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1) \ \ldots \ PT_{\overline{\phi'}}(\overline{c}_n \triangleright \overline{e}_n \rightarrow \overline{e}'_n)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$ *of depth* d.

Proof. The proof is by induction on the depth of the $\overline{\phi'}$-tree.

Induction Hypothesis: Lemma holds for $n < d$.

Induction Step: we give the interesting steps,only:

... $\overset{4.3.13}{\Longrightarrow} \exists\theta.\exists r' = (B', c \triangleright [X, e] \rightarrow [X, e']) \in \phi'$

The rule r' resulted from transforming a rule $r \in \phi$, i.e., $T_A(r) = r'$ where $r = (B, c \triangleright e \rightarrow e')$. The depth of every $\overline{\phi'}$-tree $PT_{\overline{\phi'}}(\overline{c}_k \triangleright [\overline{s^*}, \overline{e}_k] \rightarrow [\overline{s^*}, \overline{e}'_k])$ is smaller or equal to n. By the induction hypothesis there exist $\overline{\phi}$-trees $PT_{\overline{\phi}}(\overline{c}_k \triangleright \overline{e}_k \rightarrow \overline{e}'_k)$ with the same depth.

$\overset{4.3.8}{\Longrightarrow} \dfrac{\mathrm{PT}_{\overline{\phi}}(\overline{\mathrm{c}}_1 \triangleright \overline{\mathrm{e}}_1 \rightarrow \overline{\mathrm{e}}'_1)\ \ldots\ \mathrm{PT}_{\overline{\phi}}(\overline{\mathrm{c}}_\mathrm{m} \triangleright \overline{\mathrm{e}}_\mathrm{m} \rightarrow \overline{\mathrm{e}}'_\mathrm{m})}{\eta_\sigma(\theta(\mathrm{c} \triangleright \mathrm{e} \rightarrow \mathrm{e}')))}$ is a $\overline{\phi}$-tree of depth d

...

□

Finally by lemma 9.3.2 and 9.3.3 and theorem 4.3.9 follows theorem 9.3.1.

9.3.2 Correctness of Factorization

In the sequel we mean by factorization the transformation defined in Section 5.3.3. To make a set of rules deterministic, factorization has to be applied to the resulting rules as long as there are determinate rules, which are nondeterministic. We prove four properties of the transformation:

- factorization preserves determinacy
- repeated application of factorization terminates
- the resulting rules are deterministic
- factorization is correct

We assume that the rules are determinate and well-ordered.

Determinacy:

We factorize the largest set of conflicting rules with the left-hand side c, e in their conclusions.

(†) These rules are replaced by a single rule, which has the same left-hand side c, e in its conclusion and is now the only such rule and as a consequence cannot destroy determinacy.

Now we have to show, that the rules defining the new instruction are also determinate. If two rules in the original set have α-equivalent LHS in their conclusions, then there are two cases for the discrimination index j' of the first non α-equivalent LHS in their preconditions: if $j' = j$ (where j is the index used to factorize all rules with α-equivalent LHS in their preconditions), then the LHS of the conclusions of the rules defining the new instruction are not α-equivalent, otherwise if $j' > j$ then the j'-th precondition is a precondition in the resulting rules defining the new instruction and makes them determinate.

Termination:

Factorization replaces a set of conflicting rules with respect to a left-hand side c, e by a new rule using a new instruction and a set of rules defining this new instruction. Thus the number of rules with the left-hand side c, e is reduced and factorizing rules with the left-hand side c, e stops if there are no more conflicting rules or there is only one rule left.

It remains to show, that factorization of the rules defining the new instruction also terminates. The point is, that if we factorize n conflicting rules, the newly defined instruction has at most $n-1$ conflicting rules, because in condition 5.1 for the common segment j is chosen, such that $\exists k, l : k \neq l \textit{ and } e_{lj}\theta \neq e_{kj}\theta$. Thus again, factorization stops if there are no conflicting rules for the newly defined instruction or the instruction is defined by a single rule.

Deterministic Rules:

Repeated application of factorization stops, if there are no more conflicting rules. If there are no more conflicting rules, the resulting rules are deterministic.

Correctness:

In the sequel we denote by T_F one application of the factorization transformation.

Theorem 9.3.4 *Let ϕ be a set of* 2BIG *rules and $\phi' = T_F(\phi)$.*
Then $c \triangleright e \rightarrow e' \in \mathcal{I}(\overline{\phi}) \Leftrightarrow c \triangleright e \rightarrow e' \in \mathcal{I}(\overline{\phi'})$ where c, e and e' do not contain any symbols introduced by factorization.

In the proofs the notation $\check{\exists}\theta : P(\theta)$ means, there exists a smallest substitution θ which satisfies $P(\theta)$. For two substitutions θ, τ, we define the smaller relation as $\theta \leq \tau \Leftrightarrow |dom(\theta)| \leq |dom(\tau)|$. By considering the smallest substitutions in the proofs, we will not have to restrict substitutions to some set of variables. The use of smallest substitutions facilitates composition of substitutions.

Lemma 9.3.5 $PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \Rightarrow PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}')$

The proof idea is to construct from a proof tree $\frac{\mathrm{p}_1 \ \cdots \ \mathrm{p}_k}{\mathrm{x}}$ a new proof tree $\dfrac{\mathrm{p}_1 \ \cdots \ \mathrm{p}_j \quad \frac{\mathrm{p}_{j+1} \ \cdots \ \mathrm{p}_k}{\mathrm{p}}}{\mathrm{x}}$, where p is a judgement which contains an instruction introduced by factorization.

Proof. The proof is by induction on the depth d of the $\overline{\phi}$-tree.

Induction Hypothesis: Lemma holds for $n < d$.

Induction Step: Given the $\overline{\phi}$-tree $PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') = \frac{\mathrm{PT}_{\overline{\phi}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1) \ \ldots \ \mathrm{PT}_{\overline{\phi}}(\overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$

$\overset{4.3.8}{\Longrightarrow} \exists \overline{r} = (\overline{B}, \overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \in \overline{\phi}$ where $\overline{B} = \{\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1, \ldots, \overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m\}$

$\overset{4.3.13}{\Longrightarrow} \check{\exists}\theta.\exists r = (B, c \triangleright e \rightarrow e') \in \phi : \eta_\sigma(\theta(r)) = \overline{r}$

where $B = \{c_1 \triangleright e_1 \rightarrow e'_1, \ldots, c_m \triangleright e_m \rightarrow e'_m\}$

$\Rightarrow$

1st case: r was not factorized, i.e., $r \in T_F(\phi) = \phi' \overset{4.3.13}{\Longrightarrow} \overline{r} \in \overline{\phi'}$

by the induction hypothesis:

$\overline{\phi'}$-trees $PT_{\overline{\phi'}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1), \ldots, PT_{\overline{\phi'}}(\overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m)$ exist.

$\overset{4.3.8}{\Longrightarrow} PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') = \frac{\mathrm{PT}_{\overline{\phi'}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1) \ \ldots \ \mathrm{PT}_{\overline{\phi'}}(\overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$

is a $\overline{\phi'}$-tree

2nd case: r was factorized, i.e., $r \in \mathcal{C}$ where $\mathcal{C} = \{r_1, \cdots, r_k\}$ is the largest set of conflicting rules in ϕ:

$$r_1 = \frac{c_{11} \triangleright e_{11} \rightarrow e'_{11} \quad \ldots \quad c_{1m_1} \triangleright e_{1m_1} \rightarrow e'_{1m_1}}{c_{10} \triangleright e_{10} \rightarrow e'_{10}}$$
$$\vdots$$
$$r_k = \frac{c_{k1} \triangleright e_{k1} \rightarrow e'_{k1} \quad \ldots \quad c_{kmn} \triangleright e_{kmn} \rightarrow e'_{kmn}}{c_{k0} \triangleright e_{k0} \rightarrow e'_{k0}}$$

Let q be the index of r in the set $\mathcal{C}$, i.e., $r = r_q$.

Let $e^\bullet$ be the common term, $e^\dagger$ a new variable, seg the common segment and κ the new instruction as defined by the transformation, then $\mathcal{C}' = T_F(\mathcal{C})$ is the set

$$r' = \frac{seg \quad c_{1j} \triangleright e_{1j} \rightarrow e^\bullet \quad \kappa \triangleright e^\bullet \rightarrow e^\dagger}{c \triangleright e \rightarrow e^\dagger}\theta$$
$$r'_1 = \frac{c_{1(j+1)} \triangleright e_{1(j+1)} \rightarrow e'_{1(j+1)} \quad \ldots \quad c_{1m_1} \triangleright e_{1m_1} \rightarrow e'_{1m_1}}{\kappa \triangleright [\mathcal{R}, e'_{1j}] \rightarrow e'_{10}}\theta$$
$$\vdots$$
$$r'_k = \frac{c_{k(j+1)} \triangleright e_{k(j+1)} \rightarrow e'_{k(j+1)} \quad \ldots \quad c_{km_k} \triangleright e_{km_k} \rightarrow e'_{km_k}}{\kappa \triangleright [\mathcal{R}, e'_{kj}] \rightarrow e'_{k0}}\theta$$

(*) For the preconditions in the common segment, we have by the induction hypothesis

$$\forall \pi \in seg : PT_{\overline{\phi}}(\eta_\sigma(\theta(\pi))) \Rightarrow PT_{\overline{\phi'}}(\eta_\sigma(\theta(\pi)))$$

For $e^\bullet$ we know by the definiton of the operation $\odot_\theta$ that there is a smallest substitution τ such that $\tau(e^\bullet) = e'_{qj}$. Furthermore, since τ maps only variables introduced by the $\odot_\theta$ operation onto terms containing none of these new variables, τ and θ are compatible, i.e., $\forall X : X \in dom(\tau)$ *and* $X \in dom(\theta) \Rightarrow \tau(X) = \theta(X)$. Thus we have that $\eta_\sigma(\tau(\theta(c_{1j} \triangleright e_{1j} \rightarrow e^\bullet))) = \overline{c}_j \triangleright \overline{e}_j \rightarrow \overline{e}'_j$ and by the induction hypothesis follows:

$$PT_{\overline{\phi}}(\overline{c}_j \triangleright \overline{e}_j \rightarrow \overline{e}'_j) \Rightarrow PT_{\overline{\phi'}}(\overline{c}_j \triangleright \overline{e}_j \rightarrow \overline{e}'_j) = PT_{\overline{\phi'}}(\eta_\sigma(\tau(\theta(c_{1j} \triangleright e_{1j} \rightarrow e^\bullet)))) \qquad (^{**})$$

Now we have to build the proof tree for the new instruction. Since $\forall 1 \leq i \leq m_q : \eta_\sigma(\theta(c_{qi} \triangleright e_{qi} \rightarrow e'_{qi})) = \overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i$, we have by the induction hypothesis: $\forall 1 \leq i \leq m_q : PT_{\overline{\phi}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i) \Rightarrow PT_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i)$
And thus using r'_q we get the proof tree:

$$PT_{\overline{\phi'}}(\theta(\kappa) \triangleright [\overline{\mathcal{R}}, \overline{e}'_j] \rightarrow \overline{e}') = \frac{\mathrm{PT}_{\overline{\phi'}}(\overline{\mathrm{c}}_{\mathrm{j+1}} \triangleright \overline{\mathrm{e}}_{\mathrm{j+1}} \rightarrow \overline{\mathrm{e}}'_{\mathrm{j+1}}) \ \ldots \ \mathrm{PT}_{\overline{\phi'}}(\overline{\mathrm{c}}_{\mathrm{k}} \triangleright \overline{\mathrm{e}}_{\mathrm{k}} \rightarrow \overline{\mathrm{e}}'_{\mathrm{k}})}{\theta(\kappa) \triangleright [\overline{\mathcal{R}}, \overline{\mathrm{e}}'_{\mathrm{j}}] \rightarrow \overline{\mathrm{e}}'} \qquad (^{***})$$

Finally by $(^*)$,$(^{**})$,$(^{***})$ and using r':

$$PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') = \frac{\mathrm{PT}_{\overline{\phi'}}(\overline{\mathrm{c}}_1 \triangleright \overline{\mathrm{e}}_1 \rightarrow \overline{\mathrm{e}}'_1) \ \ldots \ \mathrm{PT}_{\overline{\phi'}}(\overline{\mathrm{c}}_{\mathrm{j}} \triangleright \overline{\mathrm{e}}_{\mathrm{j}} \rightarrow \overline{\mathrm{e}}'_{\mathrm{j}}) \ \ \mathrm{PT}_{\overline{\phi'}}(\theta(\kappa) \triangleright [\overline{\mathcal{R}}, \overline{\mathrm{e}}'_{\mathrm{j}}] \rightarrow \overline{\mathrm{e}}')}{\overline{\mathrm{c}} \triangleright \overline{\mathrm{e}} \rightarrow \overline{\mathrm{e}}'}$$

□

Lemma 9.3.6 $PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \Rightarrow PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}')$ *where* $\overline{c}$ *does not contain a symbol introduced by factorization.*

Proof. The proof is by induction on the depth d of the $\overline{\phi'}$-tree.

Induction Hypothesis: Lemma holds for $n < d$.

Induction Step:

1st case: $PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') = \frac{PT_{\overline{\phi'}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1) \ \ldots \ PT_{\overline{\phi'}}(\overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$ and $\overline{c}_m$ does not contain an instruction symbol introduced by factorization.
$\overset{4.3.8}{\Longrightarrow} \exists \overline{r} = (\overline{B}, \overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \in \overline{\phi'}$ where $\overline{B} = \{\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1, \ldots, \overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m\}$
$\overset{4.3.13}{\Longrightarrow} \check{\exists}\theta.\exists r = (B, c \triangleright e \rightarrow e') \in \phi' : \eta_\sigma(\theta(r)) = \overline{r}$
$\Rightarrow r$ was not factorized, i.e., $r \in \phi \Rightarrow \overline{r} \in \overline{\phi}$
Furthermore by the induction hypothesis we have:

$$\forall 1 \leq i \leq m : PT_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i) \Rightarrow PT_{\overline{\phi}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i) \qquad (^*)$$

$\overset{4.3.8}{\Longrightarrow}$ Using $r \in \phi$ and (*) we get:
$PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \to \overline{e}') = \frac{PT_{\overline{\phi}}(\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1) \ \dots \ PT_{\overline{\phi}}(\overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \to \overline{e}'}$

2nd case: $PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \to \overline{e}') = \frac{PT_{\overline{\phi'}}(\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1) \ \dots \ PT_{\overline{\phi'}}(\overline{c}_j \triangleright \overline{e}_j \to \overline{e}'_j) \ PT_{\overline{\phi'}}(\overline{\kappa} \triangleright [\overline{\mathcal{R}}, \overline{e}'_j] \to \overline{e}')}{\overline{c} \triangleright \overline{e} \to \overline{e}'}$ where $\kappa = \iota(\dots)$ for some instruction ι introduced by factorization and none of the c_i contains a symbol introduced by factorization.
$\overset{4.3.8}{\Longrightarrow} \exists \overline{r}_1 = (\overline{B_1}, \overline{c} \triangleright \overline{e} \to \overline{e}') \in \overline{\phi'}$ where $\overline{B_1} = \{\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1, \dots, \overline{c}_j \triangleright \overline{e}_j \to \overline{e}'_j, \overline{\kappa} \triangleright [\overline{\mathcal{R}}, \overline{e}'_j] \to \overline{e}'\}$
$\overset{4.3.13}{\Longrightarrow} \check{\exists}\theta.\exists r_1 = (B_1, c \triangleright e \to e') \in \phi' : \eta_\sigma(\theta(r_1)) = \overline{r_1}$
The proof tree for $\overline{\kappa} \triangleright [\overline{\mathcal{R}}, \overline{e}'_j] \to \overline{e}'$ must be of the form:
$PT_{\overline{\phi'}}(\overline{\kappa} \triangleright [\overline{\mathcal{R}}, \overline{e}'_j] \to \overline{e}') = \frac{PT_{\overline{\phi'}}(\overline{c}_{j+1} \triangleright \overline{e}_{j+1} \to \overline{e}'_{j+1}) \ \dots \ PT_{\overline{\phi'}}(\overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m)}{\overline{\kappa} \triangleright [\overline{\mathcal{R}}, \overline{e}'_j] \to \overline{e}'}$
$\overset{4.3.8}{\Longrightarrow} \exists \overline{r}_2 = (\overline{B_2}, \overline{\kappa} \triangleright [\overline{\mathcal{R}}, \overline{e}'_j] \to \overline{e}') \in \overline{\phi'}$ where $\overline{B_2} = \{\overline{c}_{j+1} \triangleright \overline{e}_{j+1} \to \overline{e}'_{j+1}, \dots, \overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m\}$
$\overset{4.3.13}{\Longrightarrow} \check{\exists}\tau.\exists r_2 = (B_2, \kappa \triangleright [R, e'_j] \to e') \in \phi' : \eta_\sigma(\tau(r_2)) = \overline{r_2}$
Since r_1 and $r_2 \in \phi'$ there must have been a rule $r_0 \in \phi$ with $r_0 = (B_0, c \triangleright e \to e')$ where $B_0 = \{c_1 \triangleright e_1 \to e'_1, \dots, c_m \triangleright e_m \to e'_m\}$
and $\eta_\sigma(\tau(\theta(r_0))) = (\{\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1, \dots, \overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m\}, \overline{c} \triangleright \overline{e} \to \overline{e}')$
Note, that τ and θ are compatible, because $\forall X \in dom(\tau) \cap dom(\theta) : X$ is a parameter of κ, i.e., $\kappa = \iota(\dots, X, \dots)$, or X occurs in the list $\mathcal{R}$, and $\tau(\kappa) = \theta(\kappa) = \overline{\kappa}$ and $\tau(\mathcal{R}) = \theta(\mathcal{R}) = \overline{\mathcal{R}}$. Furthermore by the induction hypothesis we have:

(**) $$\forall 1 \le i \le m : PT_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \to \overline{e}'_i) \Rightarrow PT_{\overline{\phi}}(\overline{c}_i \triangleright \overline{e}_i \to \overline{e}'_i)$$

$\overset{4.3.8}{\Longrightarrow}$ Using r_0 and (**) we get:
$PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \to \overline{e}') = \frac{PT_{\overline{\phi}}(\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1) \ \dots \ PT_{\overline{\phi}}(\overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \to \overline{e}'}$

□

Finally by lemma 9.3.5 and 9.3.6 and theorem 4.3.9 follows theorem 9.3.4.

9.3.3 Correctness of Removing Variables from Instructions

In the sequel we denote by T_R the transformation, which removes in the instructions of the preconditions those variables first defined in a precondition.

Theorem 9.3.7 *Let ϕ be a set of* 2BIG *rules and $\phi' = T_R(\phi)$.*
Then $c \triangleright e \to e' \in \mathcal{I}(\overline{\phi}) \Leftrightarrow c \triangleright e \to e' \in \mathcal{I}(\overline{\phi'})$ where c does not contain a symbol introduced by the transformation.

Lemma 9.3.8 $PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \to \overline{e}') \Rightarrow PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \to \overline{e}')$

Proof. The proof is by induction on the depth d of the $\overline{\phi}$-tree.

Induction Hypothesis: Lemma holds for $n < d$.

Induction Step $PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \to \overline{e}') = \frac{\mathrm{PT}_{\overline{\phi}}(\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1)\ \dots\ \mathrm{PT}_{\overline{\phi}}(\overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \to \overline{e}'}$

$\overset{4.3.8}{\Longrightarrow} \exists \overline{r} = (\overline{B}, \overline{c} \triangleright \overline{e} \to \overline{e}') \in \overline{\phi}$ where $\overline{B} = \{\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1, \dots, \overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m\}$

$\overset{4.3.13}{\Longrightarrow} \check{\exists}\theta.\exists r = (B, c \triangleright e \to e') \in \phi : \eta_\sigma(\theta(r)) = \overline{r}$

where $B = \{c_1 \triangleright e_1 \to e'_1, \dots, c_m \triangleright e_m \to e'_m\}$

$\Rightarrow \tilde{r} = (\tilde{B}, c \triangleright e \to e') \in \phi'$ where $\tilde{B} = \{\tilde{c}_1 \triangleright \tilde{e}_1 \to \tilde{e}'_1, \dots, \tilde{c}_m \triangleright \tilde{e}_m \to \tilde{e}'_m\}$

and $\tilde{c}_i, \tilde{e}_i = \begin{cases} p_i, e'_{i-1} & if\ M_i \neq \emptyset \\ c_i, e_i & otherwise \end{cases}$

Furthermore for every new instruction p_i there is a rule $\tilde{r}_i = (\{c_i \triangleright e_i \to e'_i\}, p_i \triangleright e'_{i-1} \to e'_i)$ in ϕ'.

$\Rightarrow$ Let $\theta(p_i) = \overline{p}_i$, then by the induction hypothesis the proof trees $PT_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \to \overline{e}'_i)$ exist and thus using $\tilde{r}_i$ we get $PT_{\overline{\phi'}}(\overline{p}_i \triangleright \overline{e}'_{i-1} \to \overline{e}'_i) = \frac{\mathrm{PT}_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \to \overline{e}'_i)}{\overline{p}_i \triangleright \overline{e}'_{i-1} \to \overline{e}'_i}$

$\Rightarrow PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \to \overline{e}') = \frac{pt_1\ \dots\ pt_m}{\overline{c} \triangleright \overline{e} \to \overline{e}'}$

where $pt_i = \begin{cases} PT_{\overline{\phi'}}(\overline{p}_i \triangleright \overline{e}'_{i-1} \to \overline{e}'_i) & if\ M_i \neq \emptyset \\ PT_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \to \overline{e}'_i) & otherwise \end{cases}$

□

Lemma 9.3.9 $PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \to \overline{e}') \Rightarrow PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \to \overline{e}')$

Proof. The proof is by induction on the depth d of the $\overline{\phi'}$-tree.

Induction Hypothesis: Lemma holds for $n < d$.

Induction Step: $PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') = \dfrac{PT_{\overline{\phi'}}(\overline{\tilde{c}}_1 \triangleright \overline{\tilde{e}}_1 \rightarrow \overline{e}'_1) \ \dots \ PT_{\overline{\phi'}}(\overline{\tilde{c}}_m \triangleright \overline{\tilde{e}}_m \rightarrow \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$

$\overset{4.3.8}{\Longrightarrow} \exists \overline{r} = (\overline{B}, \overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \in \overline{\phi'}$ where $\overline{B} = \{\overline{\tilde{c}}_1 \triangleright \overline{\tilde{e}}_1 \rightarrow \overline{e}'_1, \dots, \overline{\tilde{c}}_m \triangleright \overline{\tilde{e}}_m \rightarrow \overline{e}'_m\}$

$\overset{4.3.13}{\Longrightarrow} \check{\exists}\theta.\exists r = (B, c \triangleright e \rightarrow e') \in \phi' : \eta_\sigma(\theta(r)) = \overline{r}$

where $B = \{\tilde{c}_1 \triangleright \tilde{e}_1 \rightarrow e'_1, \dots, \tilde{c}_m \triangleright \tilde{e}_m \rightarrow e'_m\}$

$\Rightarrow$

1st case: $\tilde{c}_i, \tilde{e}_i = p_i, e'_{i-1}$ and $p_i = \iota_i(x_1, \dots, x_k)$ where ι_i is an instruction symbol introduced by the transformation.

$\Rightarrow \exists r_i \in \phi' : r_i = (\{c_i \triangleright e_i \rightarrow e'_i\}, p_i \triangleright e'_{i-1} \rightarrow e'_i)$

$\Rightarrow$ Let $\overline{p}_i = \theta(p_i)$ then $PT_{\overline{\phi'}}(\overline{\tilde{c}}_i \triangleright \overline{\tilde{e}}_i \rightarrow \overline{e}'_i) = PT_{\overline{\phi'}}(\overline{p}_i \triangleright \overline{e}'_{i-1} \rightarrow \overline{e}'_i) = \dfrac{PT_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i)}{\overline{p}_i \triangleright \overline{e}'_{i-1} \rightarrow \overline{e}'_i}$

Now by the induction hypothesis follows that $PT_{\overline{\phi}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i)$ exists.

2nd case: $\tilde{c}_i, \tilde{e}_i = c_i, e_i$

$\Rightarrow PT_{\overline{\phi'}}(\overline{\tilde{c}}_i \triangleright \overline{\tilde{e}}_i \rightarrow \overline{e}'_i) = PT_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i)$

and by the induction hypothesis follows that $PT_{\overline{\phi}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i)$ exists

$\overset{4.3.8}{\Longrightarrow}$ Using the original rule in ϕ we get:

$PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') = \dfrac{PT_{\overline{\phi}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1) \ \dots \ PT_{\overline{\phi}}(\overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$

□

Finally by lemma 9.3.8 and 9.3.9 and theorem 4.3.9 follows theorem 9.3.7.

9.3.4 Correctness of Sequentialization

In the sequel, we denote by T_S the sequentialization transformation.

Theorem 9.3.10 *Let ϕ be a set of* 2BIG *rules and $\phi' = T_S(\phi)$.*
Then $c \triangleright e \rightarrow e' \in \mathcal{I}(\overline{\phi}) \Leftrightarrow c \triangleright e \rightarrow e' \in \mathcal{I}(\overline{\phi'})$ where c does not contain a symbol introduced by sequentialization.

Lemma 9.3.11 $PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \Rightarrow PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}')$

Proof. The proof is by induction on the depth d of the $\overline{\phi}$-tree.

Induction Hypothesis: Lemma holds for $n < d$.

Induction Step: $PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') = \frac{PT_{\overline{\phi'}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1) \;\ldots\; PT_{\overline{\phi'}}(\overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$

$\overset{4.3.8}{\Longrightarrow} \exists \overline{r} = (\overline{B}, \overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \in \overline{\phi'}$ where $\overline{B} = \{\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1, \ldots, \overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m\}$

$\overset{4.3.13}{\Longrightarrow} \check{\exists}\theta.\exists r = (B, c \triangleright e \rightarrow e') \in \phi' : \eta_\sigma(\theta(r)) = \overline{r}$

$r = (B, c \triangleright e \rightarrow e')$ and $B = \{c_1 \triangleright e_1 \rightarrow e'_1, \ldots, c_m \triangleright e_m \rightarrow e'_m\}$

$\Rightarrow \exists r' \in \phi' : r' = (B', c \triangleright e \rightarrow e')$ and $B' = \{c_1 \triangleright e_1 \rightarrow e'_1, p_1 \triangleright e'_1 \rightarrow e_2, \ldots, p_{m-1} \triangleright e'_{m-1} \rightarrow e_m, c_m \triangleright e_m \rightarrow e'_m, p \triangleright e_m \rightarrow e'\}$

Since all variables occurring in p_i and p have been present in the original rule we have that $\overline{p}_i = \eta_\sigma(\theta(p_i))$ and $\overline{p} = \eta_\sigma(\theta(p))$ are ground.

$\overset{4.3.13}{\Longrightarrow}$

$$\overline{r}' = \eta_\sigma(\theta(r')) \in \overline{\phi'} \qquad (*)$$

For all preconditions $p_i \triangleright e'_i \rightarrow e_{i+1}$ we have a rule in ϕ without a precondition and thus $\eta_\sigma(\theta(p_i \triangleright e'_i \rightarrow e_{i+1})) \in \overline{\phi'}$

$\overset{4.3.8}{\Longrightarrow}$

$$PT_{\overline{\phi'}}(\overline{p_i} \triangleright \overline{e}'_i \rightarrow \overline{e}_{i+1}) = \frac{}{\overline{p_i} \triangleright \overline{e}'_i \rightarrow \overline{e}_{i+1}} \qquad (**)$$

analogous:

$$PT_{\overline{\phi'}}(\overline{p} \triangleright \overline{e}'_m \rightarrow \overline{e}') = \frac{}{\overline{p} \triangleright \overline{e}'_m \rightarrow \overline{e}'} \qquad (***)$$

By the induction hypothesis:

$$PT_{\overline{\phi}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i) \Rightarrow PT_{\overline{\phi'}}(\overline{c}_i \triangleright \overline{e}_i \rightarrow \overline{e}'_i) \qquad (****)$$

By (*) to (****) follows:

$$PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') = \frac{\begin{array}{l} PT_{\overline{\phi'}}(\overline{c}_1 \triangleright \overline{e}_1 \rightarrow \overline{e}'_1)\; PT_{\overline{\phi'}}(\overline{p}_1 \triangleright \overline{e}'_1 \rightarrow \overline{e}_2) \\ \ldots \\ PT_{\overline{\phi'}}(\overline{p}_{m-1} \triangleright \overline{e}'_{m-1} \rightarrow \overline{e}_m)\; PT_{\overline{\phi'}}(\overline{c}_m \triangleright \overline{e}_m \rightarrow \overline{e}'_m) \\ PT_{\overline{\phi'}}(\overline{p} \triangleright \overline{e}_m \rightarrow \overline{e}') \end{array}}{\overline{c} \triangleright \overline{e} \rightarrow \overline{e}'}$$

□

Lemma 9.3.12 $PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}') \Rightarrow PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}')$

Proof. Analogous to proof above, but the proof idea is that by removing the proof trees $PT_{\overline{\phi'}}(\overline{p_i} \triangleright \overline{e}'_i \rightarrow \overline{e}_{i+1})$ and $PT_{\overline{\phi'}}(\overline{p} \triangleright \overline{e}_m \rightarrow \overline{e}')$ from $PT_{\overline{\phi'}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}')$ we get $PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \rightarrow \overline{e}')$. □

Finally by lemma 9.3.11 and 9.3.12 and theorem 4.3.9 follows theorem 9.3.10.

9.3.5 Correctness of Conversion into TRS

We prove three properties of the conversion:

- the generated rules are linear TRS rules
- the rules are non-overlapping
- the conversion into TRS is correct

TRS:
Since conclusions of 2BIG rules are linear and none of the transformations (factorization, allocation of temporary variables, removing of variables defined in preconditions and sequentialization) produces non-linear conclusions, the generated TRS rules are linear, too.
Let $l \Rightarrow r \in R$ be a generated rule, then we have to show that $\mathcal{V}(r) \subseteq \mathcal{V}(l)$.

1st case: $\langle (c; p), e \rangle \Rightarrow \langle p, e' \rangle$
From the well-orderedness of the 2BIG rule follows that every variable in e' has to be defined in c, e.

2nd case: $\langle (c; p), e \rangle \Rightarrow \langle (c_1; \ldots; c_n; p), e_1 \rangle$
Here we have to consider two cases:

- $x \in \mathcal{V}(e_1)$: From the well-orderedness of the 2BIG rule follows that x has to be defined in c, e.
- $x \in \mathcal{V}(c_i)$: After removing of variables defined in preconditions we have that $x \in \mathcal{V}(c, e)$. This property is preserved by sequentialization.

Non-Overlapping:
After factorization, the 2BIG rules are deterministic and thus no two left hand sides of the conclusions of 2BIG rules unify. None of the following transformations introduces rules with conclusions having such left hand sides. Finally when transforming the 2BIG rules into TRS rules, the left hand sides of the conclusions become the left hand sides of the TRS rules. From this follows, that no two left hand sides of TRS rules unify. It remains to show, that no left hand side unifies with a proper subterm of another left hand side or itself, but this is obvious, because $\langle ., . \rangle$ does not occur within a term, but only as the outmost constructor.

Correctness:
In the sequel we denote by T_{TRS} the conversion into term rewriting rules.

Theorem 9.3.13 *Let ϕ be a set of allocated, sequential, deterministic* 2BIG *rules and* $R = T_{TRS}(\phi)$. *Then* $c \triangleright e \to e' \in \mathcal{I}(\overline{\phi}) \Leftrightarrow \langle c; p, e\rangle \overset{*}{\Rightarrow}_R \langle p, e'\rangle$ *for all terms* p.

Lemma 9.3.14 $PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \to \overline{e}') \Rightarrow \langle \overline{c}; \overline{p}, \overline{e}\rangle \overset{*}{\Rightarrow}_R \langle \overline{p}, \overline{e}'\rangle$

The proof idea is that based on a proof tree, we construct a sequence of rewrite steps.
Proof. The proof is by induction on the depth d of the $\overline{\phi}$-tree.

Induction Hypothesis: Lemma holds for $n < d$.

Induction Step: $PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \to \overline{e}') = \dfrac{PT_{\overline{\phi}}(\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1) \;\ldots\; PT_{\overline{\phi}}(\overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m)}{\overline{c} \triangleright \overline{e} \to \overline{e}'}$

$\overset{4.3.8}{\Longrightarrow} \exists \overline{r} = (\overline{B}, \overline{c} \triangleright \overline{e} \to \overline{e}') \in \overline{\phi}$ where $\overline{B} = \{\overline{c}_1 \triangleright \overline{e}_1 \to \overline{e}'_1, \ldots, \overline{c}_m \triangleright \overline{e}_m \to \overline{e}'_m\}$

$\overset{4.3.13}{\Longrightarrow} \exists \theta . \exists r = (B, c \triangleright e \to e') \in \phi : \eta_\sigma(\theta(r)) = \overline{r}$

where $B = \{c_1 \triangleright e_1 \to e'_1, \ldots, c_m \triangleright e_m \to e'_m\}$

$\Rightarrow r' = \langle c; p, e\rangle \Rightarrow \langle c_1; \ldots; c_m; p, e_1\rangle \in R$

Furthermore by the induction hypothesis we have:

$$\forall 1 \le i \le m : PT_{\overline{\phi}}(\overline{c}_i \triangleright \overline{e}_i \to \overline{e}'_i) \Rightarrow \langle \overline{c}_i; \overline{p}_i, \overline{e}_i\rangle \overset{*}{\Rightarrow}_R \langle \overline{p}_i, \overline{e}'_i\rangle \text{ for all terms } \overline{p}_i. \qquad (*)$$

We choose $\overline{p}_i = \overline{c}_{i+1}; \ldots; \overline{c}_m; \overline{p}$.

Let $\theta' = \theta \cup \{p \mapsto \overline{p}\}$ then $\theta'(\langle c; p, e\rangle) = \langle \overline{c}; \overline{p}, \overline{e}\rangle$.

Using r' we get: $t_0 = \langle \overline{c}; \overline{p}, \overline{e}\rangle \Rightarrow_R t_1 = \eta_\sigma(\theta'(\langle c_1; \ldots; c_m; p, e_1\rangle)) = \langle \overline{c}_1; \ldots; \overline{c}_m; \overline{p}, \overline{e}_1\rangle$.

Using $(*)$:

$t_1 \overset{*}{\Rightarrow}_R t_2 = \langle \overline{c}_2; \ldots; \overline{c}_m; \overline{p}, \overline{e}'_1\rangle$.

Since r was sequential, we have $e'_1 = e_2$, and thus $t_2 = \langle \overline{c}_2; \ldots; \overline{c}_m; \overline{p}, \overline{e}_2\rangle$.

In general, for $1 \le i < m$ we have:

Using $(*)$:

$t_i \overset{*}{\Rightarrow}_R t_{i+1} = \langle(\overline{c}_{i+1}; \ldots; \overline{c}_m; \overline{p}), \overline{e}'_i\rangle$.

Since r was sequential, we have $e'_i = e_{i+1}$, and thus $t_{i+1} = \langle \overline{c}_{i+1}; \ldots; \overline{c}_m; \overline{p}, \overline{e}_{i+1}\rangle$.

Finally using $(*)$ for t_m we have: $t_m \overset{*}{\Rightarrow}_R t = \langle \overline{p}, \overline{e}'_m\rangle$.

Since r was sequential, we have $e'_m = e'$, and thus $t = \langle \overline{p}, \overline{e}'\rangle$.

□

Lemma 9.3.15 $\langle \overline{c}; \overline{p}, \overline{e}\rangle \overset{*}{\Rightarrow}_R \langle \overline{p}, \overline{e}'\rangle \Rightarrow PT_{\overline{\phi}}(\overline{c} \triangleright \overline{e} \to \overline{e}')$

The proof idea is that based on a sequence of rewrite steps, we construct a proof tree. The proof is by induction on the length l of the rewriting sequence.

Proof.

$l = 1:$ Given the rewrite step $\langle \overline{c}; \overline{p}, \overline{e}\rangle \Rightarrow_R \langle \overline{p}, \overline{e'}\rangle$
$\Rightarrow \exists\theta.\exists r \in R : r = \langle c; p, e\rangle \Rightarrow \langle p, e'\rangle$ and $\theta(\langle c; p, e\rangle) = \langle \overline{c}; \overline{p}, \overline{e}\rangle$ and $\eta_\sigma(\theta(\langle p, e'\rangle)) = \langle \overline{p}, \overline{e'}\rangle$
$\Rightarrow c \rhd e \to e' \in \phi$
$\overset{4.3.13}{\Longrightarrow} \eta_\sigma(\theta(c \rhd e \to e')) \in \overline{\phi}$
$\overset{4.3.8}{\Longrightarrow} \dfrac{}{\overline{c} \rhd \overline{e} \to \overline{e'}}$ is a $\overline{\phi}$-tree

$l = n + 1:$ Given the sequence of rewrite steps $\langle \overline{c}; \overline{p}, \overline{e}\rangle \Rightarrow_R \ldots \Rightarrow_R \langle \overline{p}, \overline{e'}\rangle$ of length $n + 1$. There are only two kinds of rules in R. Some rules rewrite the first instruction by a sequence of instructions and some rules remove the first instruction (see $l = 1$).
$\langle \overline{c}; \overline{p}, \overline{e}\rangle \Rightarrow_R \langle(\overline{c}_1; \ldots; \overline{c}_m; \overline{p}), \overline{e}_1\rangle \Rightarrow_R \ldots \Rightarrow_R \langle \overline{p}, \overline{e'}\rangle$
$\Rightarrow \exists\theta.\exists r \in R : r = \langle c; p, e\rangle \Rightarrow \langle(c_1; \ldots; c_m; p), e_1\rangle$ and $\theta(\langle c; p, e\rangle) = \langle \overline{c}; \overline{p}, \overline{e}\rangle$
and $\eta_\sigma(\theta((\langle c_1; \ldots; c_m; p), e'\rangle)) = \langle(\overline{c}_1; \ldots; \overline{c}_m; \overline{p}), \overline{e}_1\rangle$
$\Rightarrow r_{org} = \dfrac{c_1 \rhd e_1 \to e'_1 \; \ldots \; c_m \rhd e_m \to e'_m}{c \rhd e \to e'} \in \phi \overset{4.3.13}{\Longrightarrow} \overline{r}_{org} = \dfrac{\overline{c}_1 \rhd \overline{e}_1 \to \overline{e}'_1 \; \ldots \; \overline{c}_m \rhd \overline{e}_m \to \overline{e}'_m}{\overline{c} \rhd \overline{e} \to \overline{e}'} \in \overline{\phi}$
By the induction hypothesis we have:

(*) $\forall 1 \leq i \leq m : \langle(\overline{c}_i; \ldots; \overline{c}_m; \overline{p}), \overline{e}_i\rangle \overset{*}{\Rightarrow}_R \langle(\overline{c}_{i+1}; \ldots; \overline{c}_m; \overline{p}), \overline{e}'_i\rangle$ in $\leq n$ rewrite steps
$\Rightarrow PT_{\overline{\phi}}(\overline{c}_i \rhd \overline{e}_i \to \overline{e}'_i)$

$\overset{4.3.8}{\Longrightarrow}$ By (*) and using r_{org} we get:
$PT_{\overline{\phi}}(\overline{c} \rhd \overline{e} \to \overline{e'}) = \dfrac{PT_{\overline{\phi}}(\overline{c}_1 \rhd \overline{e}_1 \to \overline{e}'_1) \; \ldots \; PT_{\overline{\phi}}(\overline{c}_m \rhd \overline{e}_m \to \overline{e}'_m)}{\overline{c} \rhd \overline{e} \to \overline{e}'}$

□

Finally by lemma 9.3.14 and 9.3.15 and theorem 4.3.9 follows theorem 9.3.13.

9.3.6 Correctness Statement for Pass Separation

Pass separation works as described in Section 5.3.10. Let T_P be the pass separation converting a set of rules R into two sets R_c and R_x: $T_P(R) = (R_c, R_x)$.
Hannan defines **abstract interpreters** as a certain class of linear term rewriting systems: (Σ, R) is an abstract interpreter, such that nop, $\langle ., .\rangle$ and $';'$ are in Σ and every rule in R is of the form: $\langle a(X_1, \ldots, X_n); P, e\rangle \Rightarrow \langle c'_1; \ldots; c'_m; P, e'\rangle$ where $X_1, \ldots, X_n$ and P are variables and $c'_1, \ldots, c'_m$ and e, e' are terms which do not contain P.

Theorem 9.3.16 (Theorem 6.9 in [Han94]) *Let R be an abstract interpreter defining term rewriting system and $T_P(R) = (R_c, R_x)$. Then*

1. *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* $\langle \tilde{c}, \tilde{e}\rangle \overset{*}{\Rightarrow}_{R_x} \langle \tilde{c'}, \tilde{e'}\rangle$
2. *if* $c \overset{\dagger}{\Rightarrow}_{R_c} \tilde{c}$, $e \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e}$ *and* $\langle \tilde{c}, \tilde{e}\rangle \overset{*}{\Rightarrow}_{R_x} \langle \tilde{c'}, \tilde{e'}\rangle$ *then* $\langle c, e\rangle \overset{*}{\Rightarrow}_R \langle c', e'\rangle$ *and* $c' \overset{\dagger}{\Rightarrow}_{R_c} \tilde{c'}$, $e' \overset{\dagger}{\Rightarrow}_{R_c} \tilde{e'}$

9.3.7 Correctness of Core System

By theorems 9.3.1, 9.3.4, 9.3.7, 9.3.10, 9.3.13, and 9.3.16 follows:

Theorem 9.3.17 *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* $c \triangleright e \rightarrow e' \in \mathcal{I}(\overline{\phi})$ *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* $c \triangleright e \rightarrow e' \in \mathcal{I}(\overline{\phi})$ *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: $c \triangleright e \rightarrow e' \in \mathcal{I}(\overline{\phi})$), 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 in Section 6.2.3. These function abstractions contain instructions and these instructions have to be compiled into the instructions of the abstract machine.

Chapter 10

Concluding Remarks

In the previous chapters, we presented a system to generate compilers and abstract machines. Although this idea is not novel, our system is the first fully automatic semantics-directed compiler generator of this kind. In this chapter, we summarize our thesis and discuss some directions for future research including extensions and applications of the system.

10.1 Summary

We gave a comparison of existing work in the field of semantics-directed generation of compilers and abstract machines. So far the derivation of abstract machines has not been assisted by automated tools. In this thesis, 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.

We gave the transformations used in our system and as an example, we transformed the 2BIG specification of two toy languages, namely SIMP and Mini-ML , and a specification of Action Semantics. For Mini-ML , we got an abstract machine which is very close to the CAM, an abstract machine used for efficient implementations of ML. We pointed out that the system was used to generate a compiler and abstract machine for action notation and that 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 2BIG specifications and the term rewriting systems are interpreted in Prolog, and all transformations have been implemented in Prolog, too. Furthermore we wrote a compiler which translates the term rewriting rules of the compiler and abstract machine into SML programs, and a compiler which generates from term rewriting rules a C implementation of the abstract machine. In Table 3.3 we listed several approaches to derive abstract machines, here is an additional entry for our system:

Author	**Quality**	**Languages**	**Transformations**	**Papers**
Stephan Diehl	aut: yes corr: yes eff: -,+,2	spec: 2BIG rules obj: term rewriting systems src: Mini-ML , SIMP, Action Notation trg: code for the abstract machine (implementations of abstract machines in Prolog, SML, and C are generated)	factorization, allocation of temporary variables, sequentialization, pass separation and various optimizations	[Die96]

The generated term rewriting rules defining the compiler and abstract machine can also be used as a basis for handwriting 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 (see section 3.7). 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.

10.2 Future Work

Given a determinate 2BIG specification, our system automatically generates a compiler and an abstract machine represented as term rewriting systems. Admittedly the transformations introduce a lot of abstract machine instructions. Especially the number of conversion instructions introduced by sequentialization should be reduced by replacing similar instructions by one more general instruction, e.g., the instructions $conv_1$ and $conv_2$ defined by the rules $conv_1, [true|R] \rightarrow R$ and $conv_2, [false|R] \rightarrow R$ might be replaced by $pop, [F|R] \rightarrow R$, but this instruction would also pop values different from $true$ and $false$. 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 optimizations have to be investigated further. Such optimizations are likely to be based on global analyses. Detection of single-threatedness 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. In the partial evaluation community [JGS93] the resulting first stage is usually not called a compiler but a **generating extension**. The second stage, i.e., the abstract machine, has so far only been derived from interpreters. Therefore the question arises, whether it makes sense to generate the second stage for other algorithms and what its practical consequences are?

Appendix A

Example Action Semantics Specification

The following sections are based on the definition of the language Mini-Δ in [BMW92] and the definition of the language SPECIMEN in [Mou93].

A.1 Syntax of Mini-Δ

$Declaration:$	$DECL ::=$	$const(id(I), EXP)$	declaration of a constant
		$var(id(I), TYPE)$	declaration of a variable
		$proc(id(I), FPLIST, COM)$	declaration of a procedure
		$fun(id(I), FPLIST, EXP)$	declaration of a function
		$dseq(DECL, DECL)$	
$Commands:$	$COM ::=$	$print(EXP)$	print to standard output
		$assign(id(I), EXP)$	assignment
		$call(id(I), APLIST)$	procedure call
		$cseq(COM, COM)$	
		$let(DECL, COM)$	local declarations
		$if(EXP, COM, COM)$	conditional
		$while(EXP, COM)$	loop
$Expressions:$	$EXP ::=$	$int(N)$	integer
		$bool(B)$	truth value
		$id(I)$	identifier
		$call(id(I), APLIST)$	function call
		$if(EXP, EXP, EXP)$	conditional
		$binary(op(OP), EXP, EXP)$	binary operation
		$unary(op(OP), EXP)$	unary operation
$Operators:$	$OP \in$	$\{add, sub, mult, div, less,$ $greater, equal, and, or, not\}$	
$Formal$ $Parameters:$	$FPLIST ::=$	FP^*	
$Formal$ $Paramters:$	$FP ::=$	$var(id(I))$	call-by-reference parameter
		$fun(id(I))$	function as parameter
		$val(id(I))$	call-by-value parameter
$Actual$ $Parameters:$	$APLIST ::=$	AP^*	
$Actual$ $Parameter:$	$AP ::=$	$var(id(I))$	call-by-reference parameter
		EXP	call-by-value parameter

A.2 Semantics of Mini-Δ

- program⟦X⟧ = execute⟦X⟧.

A.2.1 Commands

- execute⟦print(E)⟧ = |evaluate⟦E⟧
  then
  |print the value.

- execute⟦assign(id(I),E)⟧ = |evaluate⟦E⟧
  then
  |store the value in the cell bound to id(I).

- execute⟦cseq(C1,C2)⟧ = |execute⟦C1⟧
  and
  |then execute⟦C2⟧.

- execute⟦let(D,C)⟧ = |furthermore elaborate⟦D⟧
  hence
  |execute⟦C⟧.

- execute⟦if(E,C1,C2)⟧ = |evaluate⟦E⟧
  then
  |execute⟦C1⟧ else execute⟦C2⟧.

- execute⟦while(E,C)⟧ = unfolding ||evaluate⟦E⟧
  |then
  |||execute⟦C⟧ and then unfold
  ||else
  |||complete.

- execute⟦call(id(I),A)⟧ = |eval_para_list⟦A⟧ `/* procedure call */`
  then
  |enact (the abstraction bound to id(I) with the value).

A.2.2 Expressions

- evaluate⟦int(I)⟧ = give num(I) .

- evaluate⟦bool(B)⟧ = give B.

- evaluate⟦id(I)⟧ = |give the value stored in the cell bound to id(I)
  or
  |give the value bound to id(I).

- evaluate⟦if(E,E1,E2)⟧ = |evaluate⟦E⟧
  then
  |evaluate⟦E1⟧ else evaluate⟦E2⟧.

- evaluate⟦call(id(I),A)⟧ = |eval_para_list⟦A⟧ `/* function call */`
  then
  |enact the abstraction bound to id(I) with the value.

- evaluate⟦binary(O,E1,E2)⟧ = ||evaluate⟦E1⟧ then give the value label #1
  |and
  ||evaluate⟦E2⟧ then give the value label #2
  then
  |apply⟦O⟧.

- evaluate⟦unary(O,E)⟧ = evaluate⟦E⟧ then apply⟦O⟧.

A.2.3 Operators

- apply⟦op(add)⟧ = give add(the value #1, the value #2).

- apply⟦op(sub)⟧ = give subtract(the value #1, the value #2).

- apply⟦op(mult)⟧ = give multiply(the value #1, the value #2).

- apply⟦op(div)⟧ = give divide(the value #1, the value #2).

- apply⟦op(less)⟧ = give less(the value #1, the value #2).

- apply⟦op(greater)⟧ = give greater(the value #1, the value #2).

- apply⟦op(equal)⟧ = give is(the value #1, the value #2).

- apply⟦op(and)⟧ = give bool_and(the value #1, the value #2).

- apply⟦op(or)⟧ = give bool_or(the value #1, the value #2).

- apply⟦op(not)⟧ = give bool_not(the value #1).

A.2.4 Declarations

- elaborate⟦const(id(I),E)⟧ =

```
|evaluate⟦E⟧
then
|bind id(I) to the value.
```

- elaborate⟦var(id(I),T)⟧ =

```
|allocate a cell[T]
then
|bind id(I) to the cell.
```

- elaborate⟦proc(id(I),FS,C)⟧ =

```
recursively bind id(I) to closure abstraction ||furthermore elaborate_formal_para_list⟦FS⟧
                                              |hence
                                              ||execute⟦C⟧.
```

- elaborate⟦fun(id(I),FS,E)⟧ =

```
recursively bind id(I) to closure abstraction ||furthermore elaborate_formal_para_list⟦FS⟧
                                              |hence
                                              ||evaluate⟦E⟧ then give the value.
```

- elaborate⟦dseq(D1,D2)⟧ =

```
|elaborate⟦D1⟧
before
|elaborate⟦D2⟧.
```

A.2.5 Evaluate Actual Parameters

- eval_para_list⟦[]⟧ = give empty-list.

- eval_para_list⟦[E]⟧ = eval_para⟦E⟧ then give list(it).

- eval_para_list⟦[E | R]⟧ =

```
||eval_para⟦E⟧ then give list(it) label #1
|and
||eval_para_list⟦R⟧ then give it label #2
then
|give concatenation(the list #1, the list #2).
```

- eval_para⟦E⟧ = evaluate⟦E⟧.

- eval_para⟦var(id(I))⟧ = give the cell bound to id(I).

A.2.6 Elaborate Formal Parameters

- elaborate_formal_para_list⟦[]⟧ = complete.

- elaborate_formal_para_list⟦[F]⟧ = |give head-of(the list)
  then
  |elaborate_formal_para⟦F⟧.

- elaborate_formal_para_list⟦[F | R]⟧ = |give head-of(the list) then elaborate_formal_para⟦F⟧
  and then
  |give tail-of(the list) then elaborate_formal_para_list⟦R⟧.

- elaborate_formal_para⟦val(id(I),T)⟧ = ||give the value label #1
  |and
  ||allocate a cell[T] then give the cell label #2
  then
  ||bind id(I) to the cell #2
  |and
  | |store the value #1 in the cell #2.

- elaborate_formal_para⟦fun(id(I))⟧ = bind id(I) to the abstraction.

- elaborate_formal_para⟦var(id(I))⟧ = bind id(I) to the cell.

Appendix B

Excerpts of a 2BIG Specification of Action Notation

Modification wrt. de Moura's specification [Mou93]:

- Rules are determinate, all conditions are made explicit, and the order of the rules doesn't matter. We added for every language construct (if necessary) an additional rule, which applies if the other rules do not succeed.
- Instead of having a test saying $O \neq$ **completed** we let O have the value **failed**.
- We split the functionality of a test, which also binds a variable, into two parts: a test and a function call (e.g., $\mathbf{member}((A, B), L)$ binds B and yields true, to split these functionalities we replace the test by $\mathbf{member}((A, \_), L)$ where _ is an anonymous variable and replace every occurrence of B by $\mathbf{lookup}(A, L)$).
- De Moura's rule for **recursively_bind** creates a cyclic binding of logical variables. We added rules for the directive facet (see action notation book by P.D. Mosses [Mos92]) and translate **recursively_bind** into an action with directive actions:
  recursively_bind k **to** y is translated into
  $\mathbf{hence}(\mathbf{furthermore}(\mathbf{indirectly\_bind}(k, \mathbf{uninitialized})),$
  $\quad\quad \mathbf{and}(\mathbf{redirect}(k, y), \mathbf{bind}(k, \mathbf{bound}(\mathbf{value}, k))))$
  There are two different kinds of transitions in the specification. For actions we have transitions of the form $A \triangleright [T, B, S, R] \rightarrow [O, T, B, S, R]$ and for yielders we have transitions of the form $Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(D)$ where A is an action, Y is a yielder, T are transients, B are bindings, S is the store, R are redirections, O is the outcome status (**failed** or **completed**) and D is a value.

B.1 Basic Operations

We use mappings to represent transients, bindings, stores and redirections. Mappings are lists of associations $x \mapsto y$, i.e., the key x is associated with the value y.

mergeable(M_1, M_2) tests whether the mappings are mergeable. The mappings M_1 and M_2 are mergeable, if there is no key which is associated to a value in both mappings. Even in the case that a key is associated to the same value in both mappings, the mappings are not mergeable.

map_merge(M_1, M_2) merges two mergeable mappings, i.e., the result is the union of the associations in both mappings.

overlay(M_1, M_2) yields the set of all associations in both mappings, except for those with keys x occurring in both mappings. In this case the result only contains the association of x in M_1.

member(A, M) tests whether the association A is contained in the mappings M.

lookup(K, M) returns the value D if **datum**(D) is associated with the key K in the mapping M.

remove_entry(K, M) returns the mapping which results from M by removing the association for the key K.

check_type(V, T) tests whether the value V is a subtype of T.

lookup_cell_type(C) returns the type of the cell C.

new_cell(S, T) returns a new cell of type T, which is not yet associated in the store S.

lookup_red(I, R) returns the value associated with the indirection I in the redirections R.

new_indirection(R) returns a new indirection, which is not yet associated in the redirections R.

expand_unfold(U, A) returns the action which results from replacing every occurrence of **unfold** in the action A by the action U.

io_print(V) prints the value V to the output stream.

atomic(V) tests whether the value V is atomic, i.e., a string, character, integer or boolean.

B.2 A 2BIG Specification of Action Notation

Most of the LaTeX-code for typesetting the rules below has been automatically generated from the internal Prolog representation of 2BIG rules used in our system.

COMPLETE

$$\frac{}{\textbf{complete} \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, [], [], S, R]} \tag{B.1}$$

FAIL

$$\frac{}{\textbf{fail} \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, [], [], S, R]} \tag{B.2}$$

AND

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B, S', R'] \rightarrow [\textbf{completed}, T'', B'', S'', R''] \\ \#1\&\#2 \end{array}}{\textbf{and}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, \#3, \#4, S'', R'']} \tag{B.3}$$

$$\text{where} \quad \begin{array}{rcl} \#1 & = & \textbf{mergeable}(T', T'') \\ \#2 & = & \textbf{mergeable}(B', B'') \\ \#3 & = & \textbf{map\_merge}(T', T'') \\ \#4 & = & \textbf{map\_merge}(B', B'') \end{array}$$

$$\frac{A_1 \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, T', B', S', R']}{\textbf{and}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, T', B', S', R']} \tag{B.4}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B, S', R'] \rightarrow [\textbf{failed}, T'', B'', S'', R''] \end{array}}{\textbf{and}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, T'', B'', S'', R'']} \tag{B.5}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B, S', R'] \rightarrow [\mathbf{completed}, T'', B'', S'', R''] \\ \neg(\#1 \& \#2) \end{array}}{\mathbf{and}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.6}$$

where $\begin{array}{lcl} \#1 & = & \mathbf{mergeable}(T', T'') \\ \#2 & = & \mathbf{mergeable}(B', B'') \end{array}$

UNFOLDING

$$\frac{\#1 \triangleright [T, B, S, R] \rightarrow [O, T', B', S', R']}{\mathbf{unfolding}(A) \triangleright [T, B, S, R] \rightarrow [O, T', B', S', R']} \tag{B.7}$$

where $\#1 \;=\; \mathbf{expand\_unfold}(\mathbf{unfolding}(A), A)$

OR

$$\frac{A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R']}{\mathbf{or}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R']} \tag{B.8}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S', R'] \\ A_2 \triangleright [T, B, S', R'] \rightarrow [O, T'', B'', S'', R''] \end{array}}{\mathbf{or}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [O, T'', B'', S'', R'']} \tag{B.9}$$

GIVE

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(D) \\ D \neq \mathbf{nothing} \end{array}}{\mathbf{give}(Y, N) \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, [N \mapsto \mathbf{datum}(D)], [], S, R]} \tag{B.10}$$

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(D) \\ D = \mathbf{nothing} \end{array}}{\mathbf{give}(Y, N) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.11}$$

CHECK

$$\frac{Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{true})}{\mathbf{check}(Y) \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, [], [], S, R]} \tag{B.12}$$

$$\frac{Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{false})}{\mathbf{check}(Y) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.13}$$

$$\frac{Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{nothing})}{\mathbf{check}(Y) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.14}$$

THEN

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T', B, S', R'] \rightarrow [\mathbf{completed}, T'', B'', S'', R''] \\ \#1 \end{array}}{\mathbf{then}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T'', \#2, S'', R'']} \tag{B.15}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{mergeable}(B', B'') \\ \#2 & = & \mathbf{map\_merge}(B', B'') \end{array}$$

$$\frac{A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T', B', S', R']}{\mathbf{then}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T', B', S', R']} \tag{B.16}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T', B, S', R'] \rightarrow [\mathbf{failed}, T'', B'', S'', R''] \end{array}}{\mathbf{then}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T'', B'', S'', R'']} \tag{B.17}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T', B, S', R'] \rightarrow [\mathbf{completed}, T'', B'', S'', R''] \\ \neg(\#1) \end{array}}{\mathbf{then}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.18}$$

where $\#1 \;=\; \mathbf{mergeable}(B', B'')$

THE

$$\frac{\#1 \& \#2}{\mathbf{the}(Q, N) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\#3)} \tag{B.19}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{member}(N \mapsto \mathbf{datum}(D), T) \\ \#2 & = & \mathbf{check\_type}(\#3, Q) \\ \#3 & = & \mathbf{lookup}(N, T) \end{array}$$

$$\frac{\neg(\#1 \& \#2)}{\mathbf{the}(Q, N) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{nothing})} \tag{B.20}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{member}(N \mapsto \mathbf{datum}(D), T) \\ \#2 & = & \mathbf{check\_type}(\#3, Q) \\ \#3 & = & \mathbf{lookup}(N, T) \end{array}$$

BIND

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(D) \\ \#1 \end{array}}{\mathbf{bind}(K, Y) \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, [], [K \mapsto \mathbf{datum}(D)], S, R]} \tag{B.21}$$

where $\#1 = \mathbf{check\_type}(D, \mathbf{bindable})$

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(D) \\ \neg(\#1) \end{array}}{\mathbf{bind}(K, Y) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.22}$$

where $\#1 = \mathbf{check\_type}(D, \mathbf{bindable})$

FURTHERMORE

$$\frac{A \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R']}{\mathbf{furthermore}(A) \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', \#1, S', R']} \tag{B.23}$$

where $\#1 = \mathbf{overlay}(B', B)$

$$\frac{A \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T', B', S', R']}{\mathbf{furthermore}(A) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T', B', S', R']} \tag{B.24}$$

HENCE

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B', S', R'] \rightarrow [\textbf{completed}, T'', B'', S'', R''] \\ \#1 \end{array}}{\textbf{hence}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, \#2, B'', S'', R'']} \tag{B.25}$$

where $\begin{array}{lcl} \#1 & = & \textbf{mergeable}(T', T'') \\ \#2 & = & \textbf{map\_merge}(T', T'') \end{array}$

$$\frac{A_1 \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, T', B', S', R']}{\textbf{hence}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, T', B', S', R']} \tag{B.26}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B', S', R'] \rightarrow [\textbf{failed}, T'', B'', S'', R''] \end{array}}{\textbf{hence}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, T'', B'', S'', R'']} \tag{B.27}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B', S', R'] \rightarrow [\textbf{completed}, T'', B'', S'', R''] \\ \neg(\#1) \end{array}}{\textbf{hence}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, [], [], S, R]} \tag{B.28}$$

where $\#1 \;=\; \textbf{mergeable}(T', T'')$

MOREOVER

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B, S', R'] \rightarrow [\textbf{completed}, T'', B'', S'', R''] \\ \#1 \end{array}}{\textbf{moreover}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, \#2, \#3, S'', R'']} \tag{B.29}$$

where $\begin{array}{lcl} \#1 & = & \textbf{mergeable}(T', T'') \\ \#2 & = & \textbf{map\_merge}(T', T'') \\ \#3 & = & \textbf{overlay}(B', B'') \end{array}$

$$\frac{A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T', B', S', R']}{\mathbf{moreover}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T', B', S', R']} \tag{B.30}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B, S', R'] \rightarrow [\mathbf{failed}, T'', B'', S'', R''] \end{array}}{\mathbf{moreover}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T'', B'', S'', R'']} \tag{B.31}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, B, S', R'] \rightarrow [\mathbf{completed}, T'', B'', S'', R''] \\ \neg(\#1) \end{array}}{\mathbf{moreover}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.32}$$

where $\#1 \;=\; \mathbf{mergeable}(T', T'')$

BEFORE

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, \#1, S', R'] \rightarrow [\mathbf{completed}, T'', B'', S'', R''] \\ \#2 \end{array}}{\mathbf{before}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, \#3, \#4, S'', R'']} \tag{B.33}$$

$$\text{where} \quad \begin{array}{lcl} \#1 & = & \mathbf{overlay}(B', B) \\ \#2 & = & \mathbf{mergeable}(T', T'') \\ \#3 & = & \mathbf{map\_merge}(T', T'') \\ \#4 & = & \mathbf{overlay}(B'', B') \end{array}$$

$$\frac{A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T', B', S', R']}{\mathbf{before}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T', B', S', R']} \tag{B.34}$$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, \#1, S', R'] \rightarrow [\mathbf{failed}, T'', B'', S'', R''] \end{array}}{\mathbf{before}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, T'', B'', S'', R'']} \tag{B.35}$$

where $\#1 = \mathbf{overlay}(B', B)$

$$\frac{\begin{array}{c} A_1 \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, T', B', S', R'] \\ A_2 \triangleright [T, \#1, S', R'] \rightarrow [\mathbf{completed}, T'', B'', S'', R''] \\ \neg(\#2) \end{array}}{\mathbf{before}(A_1, A_2) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.36}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{overlay}(B', B) \\ \#2 & = & \mathbf{mergeable}(T', T'') \end{array}$$

BOUND

$$\frac{\#1 \qquad \#2}{\mathbf{bound}(Q, K) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\#4)} \tag{B.37}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{member}(K \mapsto \mathbf{datum}(D), B) \\ \#2 & = & \mathbf{check\_type}(\#3, \mathbf{indirection}) \\ \#3 & = & \mathbf{lookup}(K, B) \\ \#4 & = & \mathbf{lookup\_red}(\#3, R) \end{array}$$

$$\frac{\#1 \qquad \neg(\#2) \qquad \#4}{\mathbf{bound}(Q, K) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\#3)} \tag{B.38}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{member}(K \mapsto \mathbf{datum}(D), B) \\ \#2 & = & \mathbf{check\_type}(\#3, \mathbf{indirection}) \\ \#3 & = & \mathbf{lookup}(K, B) \\ \#4 & = & \mathbf{check\_type}(\#3, Q) \end{array}$$

$$\frac{\neg(\#1)}{\mathbf{bound}(Q, K) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{nothing})} \tag{B.39}$$

where $\#1 = \mathbf{member}(K \mapsto \mathbf{datum}(D), B)$

$$\frac{\#1 \qquad \neg(\#2) \qquad \neg(\#4)}{\mathbf{bound}(Q,K) \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(\mathbf{nothing})} \tag{B.40}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{member}(K \mapsto \mathbf{datum}(D), B) \\ \#2 & = & \mathbf{check\_type}(\#3, \mathbf{indirection}) \\ \#3 & = & \mathbf{lookup}(K, B) \\ \#4 & = & \mathbf{check\_type}(\#3, Q) \end{array}$$

STORE

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(D_1) \\ D_1 \neq \mathbf{nothing} \\ Y_2 \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(D_2) \\ \#1\&\#2\&\#3 \end{array}}{\mathbf{store}(Y_1,Y_2) \triangleright [T,B,S,R] \rightarrow [\mathbf{completed}, [], [], \#5, R]} \tag{B.41}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{member}(D_2 \mapsto \mathbf{datum}(D_3), S) \\ \#2 & = & \mathbf{check\_type}(D_2, \mathbf{cell}(Q)) \\ \#3 & = & \mathbf{check\_type}(D_1, \#4) \\ \#4 & = & \mathbf{lookup\_cell\_type}(D_2) \\ \#5 & = & \mathbf{overlay}([D_2 \mapsto \mathbf{datum}(D_1)], S) \end{array}$$

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(D_1) \\ D_1 = \mathbf{nothing} \end{array}}{\mathbf{store}(Y_1,Y_2) \triangleright [T,B,S,R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.42}$$

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(D_1) \\ D_1 \neq \mathbf{nothing} \\ Y_2 \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(D_2) \\ \neg(\#1\&\#2\&\#3) \end{array}}{\mathbf{store}(Y_1,Y_2) \triangleright [T,B,S,R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.43}$$

where
$$\begin{array}{lcl} \#1 & = & \mathbf{member}(D_2 \mapsto \mathbf{datum}(D_3), S) \\ \#2 & = & \mathbf{check\_type}(D_2, \mathbf{cell}(Q)) \\ \#3 & = & \mathbf{check\_type}(D_1, \#4) \\ \#4 & = & \mathbf{lookup\_cell\_type}(D_2) \end{array}$$

DEALLOCATE

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(K) \\ \#1 \end{array}}{\mathbf{deallocate}(Y) \triangleright [T, B, S, R] \rightarrow [\mathbf{completed}, [], [], \#2, R]} \tag{B.44}$$

$$\text{where} \quad \begin{array}{lcl} \#1 & = & \mathbf{member}(K \mapsto \mathbf{datum}(D), S) \\ \#2 & = & \mathbf{remove\_entry}(K, S) \end{array}$$

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(K) \\ \neg(\#1) \end{array}}{\mathbf{deallocate}(Y) \triangleright [T, B, S, R] \rightarrow [\mathbf{failed}, [], [], S, R]} \tag{B.45}$$

$$\text{where} \quad \#1 \; = \; \mathbf{member}(K \mapsto \mathbf{datum}(D), S)$$

STORED

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(K) \\ \#1\&\#2 \end{array}}{\mathbf{stored}(Q, Y) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\#3)} \tag{B.46}$$

$$\text{where} \quad \begin{array}{lcl} \#1 & = & \mathbf{member}(K \mapsto \mathbf{datum}(D), S) \\ \#2 & = & \mathbf{check\_type}(\#3, Q) \\ \#3 & = & \mathbf{lookup}(K, S) \end{array}$$

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(K) \\ \neg(\#1\&\#2) \end{array}}{\mathbf{stored}(Q, Y) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{nothing})} \tag{B.47}$$

$$\text{where} \quad \begin{array}{lcl} \#1 & = & \mathbf{member}(K \mapsto \mathbf{datum}(D), S) \\ \#2 & = & \mathbf{check\_type}(\#3, Q) \\ \#3 & = & \mathbf{lookup}(K, S) \end{array}$$

ENACT

$$\frac{\begin{array}{c} Y \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{abstraction}(A',T',B')) \\ A' \triangleright [T',B',S,R] \to [O,T'',B'',S'',R''] \end{array}}{\mathbf{enact}(Y) \triangleright [T,B,S,R] \to [O,T'',B'',S'',R'']} \tag{B.48}$$

$$\frac{Y \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{nothing})}{\mathbf{enact}(Y) \triangleright [T,B,S,R] \to [\mathbf{failed},[],[],S,R]} \tag{B.49}$$

ABSTRACT

$$\frac{}{\mathbf{abstract}(A) \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{abstraction}(A,[],[]))} \tag{B.50}$$

WITH

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{abstraction}(A,T',B')) \\ T' = [] \\ Y_2 \triangleright [T,B,S,R] \to \mathbf{datum}(D) \\ D \neq \mathbf{nothing} \end{array}}{\mathbf{with}(Y_1,Y_2) \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{abstraction}(A,[\mathbf{0} \mapsto \mathbf{datum}(D)],B'))} \tag{B.51}$$

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{abstraction}(A,T',B')) \\ T' = [] \\ Y_2 \triangleright [T,B,S,R] \to \mathbf{datum}(D) \\ D = \mathbf{nothing} \end{array}}{\mathbf{with}(Y_1,Y_2) \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{nothing})} \tag{B.52}$$

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{abstraction}(A,T',B')) \\ T' \neq [] \\ Y_2 \triangleright [T,B,S,R] \to \mathbf{datum}(D) \end{array}}{\mathbf{with}(Y_1,Y_2) \triangleright [T,B,S,R] \to \mathbf{datum}(\mathbf{abstraction}(A,T',B'))} \tag{B.53}$$

CLOSURE

$$\frac{Y \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(\mathbf{abstraction}(A,T',B')) \qquad B' = []}{\mathbf{closure}(Y) \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(\mathbf{abstraction}(A,T',B))} \tag{B.54}$$

$$\frac{Y \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(\mathbf{abstraction}(A,T',B')) \qquad B' \neq []}{\mathbf{closure}(Y) \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(\mathbf{abstraction}(A,T',B'))} \tag{B.55}$$

$$\frac{Y \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(\mathbf{nothing})}{\mathbf{closure}(Y) \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(\mathbf{nothing})} \tag{B.56}$$

ELSE

$$\frac{\#1 \qquad A_1 \triangleright [[],B,S,R] \rightarrow [O,T',B',S',R']}{\mathbf{else}(A_1,A_2) \triangleright [T,B,S,R] \rightarrow [O,T',B',S',R']} \tag{B.57}$$

where $\#1 = \mathbf{member}(\mathbf{0} \mapsto \mathbf{datum}(\mathbf{true}), T)$

$$\frac{\neg(\#1) \qquad \#2 \qquad A_2 \triangleright [[],B,S,R] \rightarrow [O,T',B',S',R']}{\mathbf{else}(A_1,A_2) \triangleright [T,B,S,R] \rightarrow [O,T',B',S',R']} \tag{B.58}$$

where
$\#1 = \mathbf{member}(\mathbf{0} \mapsto \mathbf{datum}(\mathbf{true}), T)$
$\#2 = \mathbf{member}(\mathbf{0} \mapsto \mathbf{datum}(\mathbf{false}), T)$

$$\frac{\neg(\#1) \qquad \neg(\#2)}{\mathbf{else}(A_1,A_2) \triangleright [T,B,S,R] \rightarrow [\mathbf{failed},[],[],S,R]} \tag{B.59}$$

where
$\#1 = \mathbf{member}(\mathbf{0} \mapsto \mathbf{datum}(\mathbf{true}), T)$
$\#2 = \mathbf{member}(\mathbf{0} \mapsto \mathbf{datum}(\mathbf{false}), T)$

ALLOCATE

$$\frac{}{\mathbf{allocate}(\mathbf{cell}(Q)) \triangleright [T,B,S,R] \rightarrow [\mathbf{completed}, [\mathbf{0} \mapsto \mathbf{datum}(\#1)], [], \#2, R]} \quad \text{(B.60)}$$

$$\text{where} \quad \begin{array}{lcl} \#1 & = & \mathbf{new\_cell}(S,Q) \\ \#2 & = & \mathbf{map\_merge}([\#1 \mapsto \mathbf{datum}(\mathbf{uninitialized})], S) \end{array}$$

INDIRECTLY_BIND

$$\frac{Y \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(D)}{\mathbf{indirectly\_bind}(K,Y) \triangleright [T,B,S,R] \rightarrow [\mathbf{completed}, [], [K \mapsto \mathbf{datum}(\#1)], S, \#2]} \quad \text{(B.61)}$$

$$\text{where} \quad \begin{array}{lcl} \#1 & = & \mathbf{new\_indirection}(R) \\ \#2 & = & \mathbf{overlay}([\#1 \mapsto D], R) \end{array}$$

REDIRECT

$$\frac{\begin{array}{c} Y \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(D) \\ \#1 \end{array}}{\mathbf{redirect}(K,Y) \triangleright [T,B,S,R] \rightarrow [\mathbf{completed}, [], [], S, \#2]} \quad \text{(B.62)}$$

$$\text{where} \quad \begin{array}{lcl} \#1 & = & \mathbf{member}(K \mapsto D_2, B) \\ \#2 & = & \mathbf{overlay}([\#3 \mapsto D], R) \\ \#3 & = & \mathbf{lookup}(K,B) \end{array}$$

$$\frac{\begin{array}{c} Y \triangleright [T,B,S,R] \rightarrow \mathbf{datum}(D) \\ \neg(\#1) \end{array}}{\mathbf{redirect}(K,Y) \triangleright [T,B,S,R] \rightarrow [\mathbf{failed}, [], [], S, R]} \quad \text{(B.63)}$$

$$\text{where} \quad \#1 \; = \; \mathbf{member}(K \mapsto D_2, B)$$

UNDIRECT

$$\frac{\#1}{\mathbf{undirect}(K) \triangleright [T,B,S,R] \rightarrow [\mathbf{completed}, [], [], S, \#2]} \quad \text{(B.64)}$$

$$\text{where} \quad \begin{array}{lcl} \#1 & = & \mathbf{member}(K \mapsto D, B) \\ \#2 & = & \mathbf{remove\_entry}(\#3, R) \\ \#3 & = & \mathbf{lookup\_red}(K,R) \end{array}$$

$$\frac{\neg(\#1)}{\textbf{undirect}(K) \triangleright [T, B, S, R] \rightarrow [\textbf{failed}, [], [], S, R]} \tag{B.65}$$

where $\#1 = \textbf{member}(K \mapsto D, B)$

PRINT

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \textbf{datum}(D) \\ \#1 \end{array}}{\textbf{print}(Y) \triangleright [T, B, S, R] \rightarrow [\textbf{completed}, [], [], S, R]} \tag{B.66}$$

where $\#1 = \textbf{io\_print}(D)$

LIST

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \textbf{datum}(D) \\ D \neq \textbf{nothing} \end{array}}{\textbf{list}(Y) \triangleright [T, B, S, R] \rightarrow \textbf{datum}([D])} \tag{B.67}$$

$$\frac{\begin{array}{c} Y \triangleright [T, B, S, R] \rightarrow \textbf{datum}(D) \\ D = \textbf{nothing} \end{array}}{\textbf{list}(Y) \triangleright [T, B, S, R] \rightarrow \textbf{datum}(\textbf{nothing})} \tag{B.68}$$

SUBTRACT

$$\frac{\begin{array}{c} Y_1 \triangleright [T, B, S, R] \rightarrow \textbf{datum}(D_1) \\ Y_2 \triangleright [T, B, S, R] \rightarrow \textbf{datum}(D_2) \\ D_1 \neq \textbf{nothing} \& D_2 \neq \textbf{nothing} \end{array}}{\textbf{subtract}(Y_1, Y_2) \triangleright [T, B, S, R] \rightarrow \textbf{datum}(\#1)} \tag{B.69}$$

where $\#1 = D_1 - D_2$

$$\frac{\begin{array}{c} Y_1 \triangleright [T, B, S, R] \rightarrow \textbf{datum}(D_1) \\ Y_2 \triangleright [T, B, S, R] \rightarrow \textbf{datum}(D_2) \\ \neg(D_1 \neq \textbf{nothing} \& D_2 \neq \textbf{nothing}) \end{array}}{\textbf{subtract}(Y_1, Y_2) \triangleright [T, B, S, R] \rightarrow \textbf{datum}(\textbf{nothing})} \tag{B.70}$$

LESS

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \rightarrow \textbf{datum}(D_1) \\ Y_2 \triangleright [T,B,S,R] \rightarrow \textbf{datum}(D_2) \\ \#1\&\#2 \\ D_1 < D_2 \end{array}}{\textbf{less}(Y_1,Y_2) \triangleright [T,B,S,R] \rightarrow \textbf{datum}(\textbf{true})} \tag{B.71}$$

where $\begin{array}{lcl} \#1 & = & \textbf{check\_type}(D_1, \textbf{integer}) \\ \#2 & = & \textbf{check\_type}(D_2, \textbf{integer}) \end{array}$

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \rightarrow \textbf{datum}(D_1) \\ Y_2 \triangleright [T,B,S,R] \rightarrow \textbf{datum}(D_2) \\ \#1\&\#2 \\ \neg(D_1 < D_2) \end{array}}{\textbf{less}(Y_1,Y_2) \triangleright [T,B,S,R] \rightarrow \textbf{datum}(\textbf{false})} \tag{B.72}$$

where $\begin{array}{lcl} \#1 & = & \textbf{check\_type}(D_1, \textbf{integer}) \\ \#2 & = & \textbf{check\_type}(D_2, \textbf{integer}) \end{array}$

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \rightarrow \textbf{datum}(D_1) \\ Y_2 \triangleright [T,B,S,R] \rightarrow \textbf{datum}(D_2) \\ \neg(\#1\&\#2) \end{array}}{\textbf{less}(Y_1,Y_2) \triangleright [T,B,S,R] \rightarrow \textbf{datum}(\textbf{nothing})} \tag{B.73}$$

where $\begin{array}{lcl} \#1 & = & \textbf{check\_type}(D_1, \textbf{integer}) \\ \#2 & = & \textbf{check\_type}(D_2, \textbf{integer}) \end{array}$

BOOL_AND

$$\frac{\begin{array}{c} Y_1 \triangleright [T,B,S,R] \rightarrow \textbf{datum}(\textbf{true}) \\ Y_2 \triangleright [T,B,S,R] \rightarrow \textbf{datum}(\textbf{true}) \end{array}}{\textbf{bool\_and}(Y_1,Y_2) \triangleright [T,B,S,R] \rightarrow \textbf{datum}(\textbf{true})} \tag{B.74}$$

$$\frac{\begin{array}{c} Y_1 \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{true}) \\ Y_2 \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{false}) \end{array}}{\mathbf{bool\_and}(Y_1, Y_2) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{false})} \tag{B.75}$$

$$\frac{Y_1 \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{false})}{\mathbf{bool\_and}(Y_1, Y_2) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{false})} \tag{B.76}$$

NUM

$$\frac{\#1}{\mathbf{num}(N) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(N)} \tag{B.77}$$

where $\#1 \quad = \quad \mathbf{atomic}(N)$

$$\frac{\neg(\#1)}{\mathbf{num}(N) \triangleright [T, B, S, R] \rightarrow \mathbf{datum}(\mathbf{nothing})} \tag{B.78}$$

where $\#1 \quad = \quad \mathbf{atomic}(N)$

Appendix C

Transforming the 2BIG Specification of SIMP

C.1 The 2BIG Specification of SIMP

$$\frac{B \triangleright E \to \textbf{true} \quad C_1 \triangleright E \to E'}{\textbf{if}(B, C_1, C_2) \triangleright E \to E'} \qquad \frac{B \triangleright E \to \textbf{false} \quad C_2 \triangleright E \to E'}{\textbf{if}(B, C_1, C_2) \triangleright E \to E'} \tag{C.1}$$

$$\frac{C_1 \triangleright E \to E' \quad C_2 \triangleright E' \to E''}{\textbf{seq}(C_1, C_2) \triangleright E \to E''} \qquad \frac{\textbf{io\_print}(\textbf{lookup}(X, E))}{\textbf{print}(X) \triangleright E \to E} \tag{C.2}$$

$$\frac{B \triangleright E \to \textbf{true} \quad \textbf{seq}(C, \textbf{while}(B, C)) \triangleright E \to E'}{\textbf{while}(B, C) \triangleright E \to E'} \qquad \frac{B \triangleright E \to \textbf{false}}{\textbf{while}(B, C) \triangleright E \to E} \tag{C.3}$$

$$\frac{V \triangleright E \to V'}{\textbf{assign}(X, V) \triangleright E \to \textbf{replace}(X, V', E)} \tag{C.4}$$

$$\frac{}{\textbf{id\_op}(X) \triangleright E \to \textbf{lookup}(X, E)} \qquad \frac{}{\textbf{num}(N) \triangleright E \to N} \tag{C.5}$$

$$\frac{V_1 \triangleright E \to V_1' \quad V_2 \triangleright E \to V_2'}{\textbf{add}(V_1, V_2) \triangleright E \to \textbf{plus}(V_1', V_2')} \qquad \frac{V_1 \triangleright E \to V_1' \quad V_2 \triangleright E \to V_2'}{\textbf{eq}(V_1, V_2) \triangleright E \to \textbf{equal}(V_1', V_2')} \tag{C.6}$$

C.2 The generated Compiler for SIMP

$$
\begin{array}{lcl}
\mathbf{if}(B, C_1, C_2) & \Rightarrow & \overline{\mathbf{if}}; B; \overline{\mathbf{conv\_0}}; \overline{\mathbf{factor\_0}}(C_2, C_1) \\
\mathbf{seq}(C_1, C_2) & \Rightarrow & \overline{\mathbf{seq}}; C_1; C_2 \\
\mathbf{while}(B, C) & \Rightarrow & \overline{\mathbf{while}}(B, C) \\
\mathbf{assign}(X, V) & \Rightarrow & \overline{\mathbf{if}}; V; \overline{\mathbf{conv\_2}}(X) \\
\mathbf{id\_op}(X) & \Rightarrow & \overline{\mathbf{id\_op}}(X) \\
\mathbf{num}(N) & \Rightarrow & \overline{\mathbf{num}}(N) \\
\mathbf{add}(V_1, V_2) & \Rightarrow & \overline{\mathbf{if}}; V_1; \overline{\mathbf{conv\_3}}; V_2; \overline{\mathbf{conv\_4}} \\
\mathbf{eq}(V_1, V_2) & \Rightarrow & \overline{\mathbf{if}}; V_1; \overline{\mathbf{conv\_3}}; V_2; \overline{\mathbf{conv\_6}} \\
\mathbf{not}(B) & \Rightarrow & \overline{\mathbf{seq}}; B; \overline{\mathbf{conv\_7}}; \overline{\mathbf{factor\_2}} \\
\mathbf{print}(X) & \Rightarrow & \overline{\mathbf{print}}(X)
\end{array}
$$

C.3 The generated Abstract Machine for SIMP

$$
\begin{array}{lcl}
\langle \overline{\mathbf{if}}; C,\ [D, E] \rangle & \Rightarrow & \langle C,\ [[[E]|D], E] \rangle \\
\langle \overline{\mathbf{conv\_0}}; C,\ [[[E]|D], B] \rangle & \Rightarrow & \langle C,\ [D, [[E], B]] \rangle \\
\langle \overline{\mathbf{factor\_0}}(C_1, C_2); C,\ [D, [[E], \mathbf{true}]] \rangle & \Rightarrow & \langle C_2; C,\ [D, E] \rangle \\
\langle \overline{\mathbf{factor\_0}}(C_1, C_2); C,\ [D, [[E], \mathbf{false}]] \rangle & \Rightarrow & \langle C_1; C,\ [D, E] \rangle \\
\langle \overline{\mathbf{seq}}; C,\ [D, E] \rangle & \Rightarrow & \langle C,\ [D, E] \rangle \\
\langle \overline{\mathbf{while}}(B, C_1); C, [D, E] \rangle & \Rightarrow & \langle B; \overline{\mathbf{conv\_0}}; \overline{\mathbf{factor\_1}}(C_1, B); C,\ [[[E]|D], E] \rangle \\
\langle \overline{\mathbf{factor\_1}}(C_1, B); C,\ [D, [[E], \mathbf{true}]] \rangle & \Rightarrow & \langle \overline{\mathbf{seq}}; B; \overline{\mathbf{while}}(B, C_1); C,\ [D, E] \rangle \\
\langle \overline{\mathbf{factor\_1}}(C_1, B); C,\ [D, [[E], \mathbf{false}]] \rangle & \Rightarrow & \langle C,\ [D, E] \rangle \\
\langle \overline{\mathbf{conv\_2}}(X); C,\ [[[E]|D], V] \rangle & \Rightarrow & \langle C,\ [D, \mathbf{replace}(X, V, E)] \rangle \\
\langle \overline{\mathbf{id\_op}}(X); C,\ [D, E] \rangle & \Rightarrow & \langle C,\ [D, \mathbf{lookup}(E, X)] \rangle \\
\langle \overline{\mathbf{num}}(N); C,\ [D, E] \rangle & \Rightarrow & \langle C,\ [D, N] \rangle \\
\langle \overline{\mathbf{conv\_4}}; C,\ [[[V_1]|D], V_2] \rangle & \Rightarrow & \langle C,\ [D, \mathbf{plus}(V_1, V_2)] \rangle \\
\langle \overline{\mathbf{conv\_3}}; C,\ [[[E]|D], V] \rangle & \Rightarrow & \langle C,\ [[[V]|D], E] \rangle \\
\langle \overline{\mathbf{conv\_6}}; C,\ [[[V_1]|D], V_2] \rangle & \Rightarrow & \langle C,\ [D, \mathbf{equal}(V_1, V_2)] \rangle \\
\langle \overline{\mathbf{conv\_7}}; C,\ [D, E] \rangle & \Rightarrow & \langle C,\ [D, [[], E]] \rangle \\
\langle \overline{\mathbf{factor\_2}}; C,\ [D, [[], \mathbf{true}]] \rangle & \Rightarrow & \langle C,\ [D, \mathbf{false}] \rangle \\
\langle \overline{\mathbf{factor\_2}}; C,\ [D, [[], \mathbf{false}]] \rangle & \Rightarrow & \langle C,\ [D, \mathbf{true}] \rangle \\
\langle \overline{\mathbf{print}}(X); C,\ [D, E] \rangle & \Rightarrow & \langle \overline{\mathbf{test\_0}}(X); \overline{\mathbf{conv\_8}}; C,\ [[[E]|D], [E]] \rangle \\
\langle \overline{\mathbf{conv\_8}}; C,\ [[[E]|D], \mathbf{true}] \rangle & \Rightarrow & \langle C,\ [D, E] \rangle \\
\langle \overline{\mathbf{test\_0}}(X); C,\ [D, [E]] \rangle & \Rightarrow & \langle C,\ [D, \mathbf{io\_print}(\mathbf{lookup}(E, X))] \rangle
\end{array}
$$

C.4 Basic Operations

We use mappings to represent bindings of identifiers to values.

replace(X, V, B) replaces the binding for the identifier X in the binding B by a binding of X to the value V.

lookup(B, X) returns the value bound to the identifier X in the binding B.

plus(N, M) returns the sum of the integers N and M.

equal(V, W) tests whether the values V and W are equal.

io_print(V) prints the value V to the output stream.

Appendix D

A Note on Factorization

Consider the following rules:

$$\textbf{1)}\ \frac{a \triangleright e \rightarrow \textbf{completed}, e'}{a\ \textbf{or}\ b \triangleright e \rightarrow \textbf{completed}, e'}$$

$$\textbf{2)}\ \frac{a \triangleright e \rightarrow \textbf{failed}, e' \quad b \triangleright e \rightarrow \textbf{completed}, e''}{a\ \textbf{or}\ b \triangleright e \rightarrow \textbf{completed}, e''}$$

$$\textbf{3)}\ \frac{a \triangleright e \rightarrow \textbf{failed}, e' \quad b \triangleright e \rightarrow \textbf{failed}, e''}{a\ \textbf{or}\ b \triangleright e \rightarrow \textbf{failed}, e}$$

The first solution might be to use factorization for two rules several times. Factorize 2) and 3):

$$\textbf{23)}\ \frac{a \triangleright e \rightarrow \textbf{failed}, e' \quad b \triangleright e \rightarrow (o, e'') \quad \textbf{ins1}(e) \triangleright (o, e'') \rightarrow o', e^{+}}{a\ \textbf{or}\ b \triangleright e \rightarrow o', e^{+}}$$

a) $\textbf{ins1}(e) \triangleright (\textbf{completed}, e') \rightarrow \textbf{completed}, e'$

b) $\textbf{ins1}(e) \triangleright (\textbf{failed}, e') \rightarrow \textbf{failed}, e$

Now we have still two conflicting rules 1) and 23).
Let's factorize 1) and 23)

$$\textbf{123)}\ \frac{a \triangleright e \rightarrow o, e' \quad \textbf{ins2}(e, b) \triangleright (o, e') \rightarrow o', e''}{a\ \textbf{or}\ b \triangleright e \rightarrow o', e''}$$

c) $\textbf{ins2}(e, b) \triangleright (\textbf{completed}, e') \rightarrow \textbf{completed}, e'$

d) $$\frac{b \triangleright e \rightarrow (o,e'') \quad \mathbf{ins1}(e) \triangleright (o,e'') \rightarrow o',e^{+}}{\mathbf{ins2}(e,b) \triangleright (\mathbf{failed},e') \rightarrow o',e^{+}}$$

OK, this worked fine !
Now we start again, this time we decide to factorize 1) and 2) first.

12) $$\frac{a \triangleright e \rightarrow o,e' \quad \mathbf{ins1}(e,b) \triangleright (o,e') \rightarrow \mathbf{completed},e''}{a \textbf{ or } b \triangleright e \rightarrow \mathbf{completed},e''}$$

a) $\mathbf{ins1}(e,b) \triangleright (\mathbf{completed}, e') \rightarrow \mathbf{completed}, e'$

b) $$\frac{b \triangleright e \rightarrow \mathbf{completed},e''}{\mathbf{ins1}(e,b) \triangleright (\mathbf{failed},e') \rightarrow \mathbf{completed},e''}$$

This time 12) and 3) are the remaining conflicting rules, so we will factorize these:

123) $$\frac{a \triangleright e \rightarrow o,e' \quad \mathbf{ins2}(e,b) \triangleright (o,e') \rightarrow o',e''}{a \textbf{ or } b \triangleright e \rightarrow o',e''}$$

c) $$\frac{b \triangleright e \rightarrow \mathbf{failed},e''}{\mathbf{ins2}(e,b) \triangleright (\mathbf{failed},e') \rightarrow \mathbf{failed},e}$$

d) $$\frac{\mathbf{ins1}(e,b) \triangleright (o,e') \rightarrow \mathbf{completed},e''}{\mathbf{ins2}(e,b) \triangleright (o,e') \rightarrow \mathbf{completed},e''}$$

Now the left hand sides of the conclusions in c) and d) do not really clash, because they are not α-equal but one is more general than the other. But we would have to factorize the two of them, otherwise two rules would be applicable for $\mathbf{ins2}(e, b), (\mathbf{failed}, e')$.
Our strategy to factorize more than two rules works as follows: In a set of m rules for the same instruction, there can be several conflicting subsets. What we suggest is to factorize the largest subsets first.
So in the above example the three rules are the largest subset of conflicting rules and we factorize them as follows:

123) $$\frac{a \triangleright e \rightarrow o,e' \quad \mathbf{ins1}(b,e) \triangleright (o,e') \rightarrow o',e''}{a \textbf{ or } b \triangleright e \rightarrow o',e''}$$

a) $\mathbf{ins1}(b,e) \triangleright (\mathbf{completed}, e') \rightarrow \mathbf{completed}, e'$

b) $$\frac{b \triangleright e \rightarrow \mathbf{completed},e''}{\mathbf{ins1}(b,e) \triangleright (\mathbf{failed},e') \rightarrow \mathbf{completed},e''}$$

c) $$\frac{b \triangleright e \rightarrow \mathbf{failed}, e''}{\mathbf{ins1}(b, e) \triangleright (\mathbf{failed}, e') \rightarrow \mathbf{failed}, e}$$

Now b) and c) are conflicting rules and we factorize these:

bc) $$\frac{b \triangleright e \rightarrow o, e'' \quad \mathbf{ins2}(e) \triangleright (o, e'') \rightarrow o', e^{+}}{\mathbf{ins1}(b, e) \triangleright (\mathbf{failed}, e') \rightarrow o', e^{+}}$$

d) $\mathbf{ins2}(e) \triangleright (\mathbf{completed}, e'') \rightarrow \mathbf{completed}, e''$

e) $\mathbf{ins2}(e) \triangleright (\mathbf{failed}, e'') \rightarrow \mathbf{failed}, e$

This is similar to what we got in our first attempt, but in the first attempt there was no obvious reason to prefer the rules 2) and 3) over the rules 1) and 2). And as shown above, the factorization of the latter leads to severe problems.
A closer look at these problems reveals that the old strategy destroys determinacy. In rules 12) and 3) the terms o, e' and $\mathbf{failed}, e'$ in the first precondition are not unifiable. In the proof in Section 9.3.2 a (†) marks where our strategy was used to prove preservation of determinacy.

Bibliography

[Acz77] Peter Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, **Handbook of Mathematical Logic**. North-Holland, 1977.

[AK91] Hassan Aït-Kaci. **Warren's Abstract Machine - A Tutorial Reconstruction**. MIT Press, 1991.

[App89] Andrew W. Appel. Continuation-Passing, Closure-Passing Style. In **Proc. of POPL'89**. 1989.

[App92] Andrew W. Appel. **Compiling with Continuations**. Cambridge University Press, 1992.

[Ast91] Egidio Astesiano. Inductive and Operational Semantics. In E.J. Neuhold and M. Paul, editors, **Formal Description of Programming Concepts**. Springer Verlag, IFIP, 1991.

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

[AU72] A.V. Aho and J.D. Ullman. **The Theory of Parsing, Translation and Compiling**. Prentice-Hall, 1972.

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

[BFHW94] C. Böschen, C. Fecht, A.V. Hense, and R. Wilhelm. An Abstract Machine for an Object-Oriented Language with Top-Level Classes. Technical Report FB14 - No. A 02/94, Computer Science Department, University Saarbrücken, 1994.

[BMW92] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an Action Semantics Directed Compiler Generator. In **CC'92**, volume LNCS 641. Springer Verlag, 1992.

[BP93] Anders Bondorf and Jens Palsberg. Compiling Actions by Partial Evaluation. **FPCA'93**, 1993.

[BR92] Egon Börger and Dean Rosenzweig. The WAM – Definition and Compiler Correctness. Technical Report TR-14/92, Universita Degli Studi Di Pisa, Pisa, Italy, 1992.

[Car84] Luca Cardelli. Compiling a functional language. In **International Symposium on LISP and Functional Programming**, 1984.

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The Categorial Abstract Machine. In **Proceedings of FPCA'85**, volume LNCS 201. Springer Verlag, 1985.

[CJ83] Henning Christiansen and Neil Jones. Control Flow Treatment in a Simple Semantics-Directed Compiler Generator. In D. Bjørner, editor, **Formal Description of Programming Concepts**, volume II. North-Holland, 1983.

[CK91] C. Consel and S.C. Khoo. Semantics-Directed Generation of a Prolog Compiler. In **Springer Verlag**, volume LNCS 528, pages 135–146. 1991.

[Des84] Thierry Despeyroux. Executable Specification of Static Semantics. In **Semantics of Data Types**, volume LNCS 173. Springer Verlag, 1984.

[Des86] Joëlle Despeyroux. Proof of Translation in Natural Semantics. In **First Symposium on Logic in Computer Science, LICS'86**, volume LNCS 213. Springer Verlag, 1986.

[Diea] Stephan Diehl. A Prolog Positive Supercompiler. unpublished draft.

[Dieb] Stephan Diehl. An Experiment in Abstract Machine Design. (to appear in "Software – Practice and Experience").

[Die93] Stephan Diehl. Prolog and Typed Feature Structures: A Compiler for Parallel Computers. Master's thesis, Worcester Polytechnic Institute, Worcester, Massachusetts, 1993.

[Die95a] Stephan Diehl. Automatic Generation of a Compiler and an Abstract Machine for Action Notation. Technical Report FB14 No. A-03/95, University Saarbrücken, 1995.

[Die95b] Stephan Diehl. Transformations of Evolving Algebras. Technical Report FB14 No. A-02/95, University Saarbrücken, 1995.

[Die96] Stephan Diehl. **Semantics-Directed Generation of Compilers and Abstract Machines**. PhD thesis, University Saarbrücken, Germany, 1996.

[DJ86] Mads Dam and Frank Jensen. Compiler Generation from Relational Semantics. In **ESOP'86**, volume LNCS 213. Springer Verlag, 1986.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho, editors. **Attribute Grammars – Definitions, Systems and Bibliography**, volume LNCS 323. Springer Verlag, 1988.

[DM82] L. Damas and R. Milner. Principal Type-Schemes for Functional Languages. In **ACM Symposium on Principles of Programming Languages**. 1982.

[dS90] Fabio Q.B. da Silva. Towards a Formal Framework for Evaluation of Operational Semantics. Technical Report ECS-LFCS-90-126, Edinburgh University, 1990.

[dS92] Fabio Q.B. da Silva. **Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics**. PhD thesis, University of Edinburgh, 1992.

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

[Gau81] M.C. Gaudel. Compiler Generation from Formal Definition of Programming Languages: A Survey. volume LNCS 107. Springer Verlag, 1981.

[Gau83] M.C. Gaudel. Correctness Proof of Programming Language Translation. In D. Bjørner, editor, **Formal Description of Programming Concepts**, volume II. North-Holland, 1983.

[GRW77] H. Ganzinger, K. Ripken, and R. Wilhelm. Automatic Generation of Optimizing Multipass Compilers. In B. Gilchrist, editor, **Information Processing 77**. North-Holland, 1977.

[GS94] R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In **PLILP'94**. 1994.

[Gur91] Yuri Gurevich. Evolving Algebras: a tutorial introduction. **Bulletin of the European Association for Theoretical Computer Science**, 43:264–284, 1991.

[Han91a] J. Hannan. Staging transformations for abstract machines. In **Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)**, pages 130–141. New York: ACM, 1991.

[Han91b] John Hannan. Making Abstract Machines Less Abstract. In **Proc. of FPCA'91**, volume LNCS 523, pages 618–635. Springer Verlag, 1991.

[Han94] John Hannan. Operational Semantics-Directed Compilers and Machine Architectures. **ACM TOPLAS**, 16(4), 1994.

[Has88] Laurent Hascoët. Partial Evaluation with Inference Rules. **New Generation Computing**, 6, 1988.

[HM90] John Hannan and Dale Miller. From Operational Semantics to Abstract Machines: Preliminary Results. In **Proc. of the ACM Conference on Lisp and Functional Programming**. 1990.

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

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

[Joh84] T. Johnson. Efficient Compilation of Lazy Evaluation. In **CC'84**. Sigplan Notices 19(6), 1984.

[Jon80] N.D. Jones, editor. **Semantics Directed Compiler Generation**, volume LNCS 94. Springer Verlag, 1980.

[Jon88] N. D. Jones. Challenging problems in partial evaluation and mixed computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, **Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987**. North-Holland, 1988.

[Jou95] Jean-Pierre Jouannaud. Introduction to Rewriting. In Hubert Comon and Jean-Pierre Jouannaud, editors, **Term Rewriting**, volume LNCS 909. Springer Verlag, 1995.

[JS86] U. Jørring and W.L. Scherlis. Compilers and staging transformations. In **Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida**, pages 86–96. New York: ACM, 1986.

[JSMY92] J. Jaffar, P.J. Stuckey, S. Michaylov, and R.H.C. Yap. An Abstract Machine for CLP(R). In **PLDI'92, San Francisco**. Sigplan Notices, 1992.

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

[Kel89] Richard A. Kelsey. **Compilation by Program Transformation**. PhD thesis, Yale University, 1989.

[Kel95] Richard A. Kelsey. A Correspondence between Continuation Passing Style and Static Single Assignment Form. **Sigplan Notices**, to appear, 1995.

[KH89] R.A. Kelsey and P. Hudak. Realistic compilation by program transformation : detailed summary. In **Proc. of POPL'89**. 1989.

[KHZ82] U. Karstens, B. Hutt, and E. Zimmermann. GAG: A Practical Compiler Generator. volume LNCS 141. Springer Verlag, 1982.

[Knu68] D. Knuth. Semantics of Context-Free Languages. **Math. Systems Theory**, 2, 1968.

[Kur86] P. Kursawe. How to invent a Prolog machine. In **Proc. Third International Conference on Logic Programming**, pages 134–148. Springer LNCS 225, 1986.

[Kur87] P. Kursawe. How to invent a Prolog machine. **New Generation Computing**, 5:97–114, 1987.

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

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

[Lem92] Karen A. Lemone. **Design of Compilers: techniques of programming language translation**. CRC Press, 1992.

[Ler93] X. Leroy. The Caml Light System, release 0.6 - documentation and user's manual. Technical report, INRIA, France, 1993.

[LHPT94] Paul Lukowicz, Ernst A. Heinz, Lutz Prechelt, and Walter F. Tichy. Experimental Evaluation in Computer Science: A Quantitative Study. Technical Report 17/94 (to appear in "Journal of Systems and Software"), University of Karlsruhe, Germany, 1994.

[Llo87] J.W. Lloyd. **Foundations of Logic Programming**. Springer Verlag, second extended edition, 1987.

[LMW86] J. Loeckx, K. Mehlhorn, and R. Wilhelm. **Grundlagen der Programmiersprachen**. Teubner Verlag, 1986.

[LP81] H.R. Lewis and C.H. Papadimitriou. **Elements of the Theory of Computation**. Prentice-Hall, 1981.

[McK94] Stephen McKeever. A Framework for Generating Compilers from Natural Semantics Specifications. In P.D. Mosses, editor, **Proc. of the 1st Workshop on Action Semantics**, BRICS-NS-94-1. University of Aarhus, Denmark, 1994.

[Mey90] B. Meyer. **Introduction to the Theory of Programming Languages**. Prentice Hall, 1990.

[Mic95] Sun Microsystems. **Documentation of the Java Developers Kit – version 1.0 Beta**, 1995. available at http://java.sun.com/JDK-beta/index.html.

[MLB76] M. Marcotty, H.F. Legard, and G.V. Bochmann. A Sampler of Formal Definitions. **ACM Computing Surveys**, 8(2), 1976.

[Mos83] Peter Mosses. Abstract Semantic Algebras! In D. Bjørner, editor, **Formal Description of Programming Concepts**, volume II. North-Holland, 1983.

[Mos84] Peter Mosses. A Basic Abstract Semantic Algebra. In **Semantics of Data Types**, volume LNCS 173. Springer Verlag, 1984.

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

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

[MS86] M. Mauny and A. Suarez. Implementing functional languages in the categorial abstract machine. In **International Conference on LISP and Functional Programming**, 1986.

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

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

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

[MW94] H. Moura and D. A. Watt. Action Transformations in the ACTRESS Compiler Generator. In **CC'94**, volume LNCS 768. Springer Verlag, 1994.

[Nil92] Ulf Nilsson. **Abstract Interpretation & Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs**. PhD thesis, Linköping University, Sweden, 1992.

[Nil93] Ulf Nilsson. Towards a methodology for the design of abstract machines. **Journal of Logic Programming**, 16:163–189, 1993.

[NN86] F. Nielson and H.R. Nielson. Code Generation from Two-Level Denotational Meta-Languages. In H. Ganzinger, N.D. Jones, editor, **Programs as Data Objects**, volume LNCS 217. Springer Verlag, 1986.

[NN92] H.R. Nielson and F. Nielson. **Semantics with Applications – A Formal Introduction**. John Wiley & Sons Ltd., 1992.

[Orb94] Peter Orbæk. OASIS: An Optimizing Action-Based Compiler Generator. In **CC'94**, volume LNCS 768. Springer Verlag, 1994.

[Pal92a] Jens Palsberg. A provably correct Compiler Generator. In **ESOP'92**, volume LNCS 582. Springer Verlag, 1992.

[Pal92b] Jens Palsberg. An Automatically Generated and Provably Correct Compiler for a Subset of Ada. DAIMI PB 383, Computer Science Department, Aarhus University, 1992.

[Pau82] L. Paulson. A Semantics-Directed Compiler Generator. In **ACM Symposium on Principles of Programming Languages**. 1982.

[PD82] St. Pemberton and M. Daniels. **Pascal Implementation, The P4 Compiler**. Ellis Horwood, 1982.

[Pet94] Mikael Pettersson. RML - a new language and implementation for natural semantics. In M. Hermenegildo and J. Penjam, editors, **Proc. of the 6th Int. Symp. on Programming Language Implementation and Logic Programming, PLILP'94**, volume LNCS 844. Springer Verlag, 1994.

[Pet95] Mikael Pettersson. **Compiling Natural Semantics**. PhD thesis, Linköping University, Sweden, 1995.

[PL87] U.F. Pleban and P. Lee. A Realistic Compiler based on High-Level Semantics: Another Progress Report. In **Proc. of POPL'87**. 1987.

[PL88] U.F. Pleban and P. Lee. High-Level Semantics. In **Springer Verlag**, volume LNCS 298. 1988.

[Plo81] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN 19, Computer Science Department, Aarhus University, 1981.

[Räi80] K. Räihä. Experience with the Compiler Writing System HLP. volume LNCS 94. Springer Verlag, 1980.

[Ras82] Martin R. Raskovsky. Denotational Semantics as a Specification of Code Generators. **Sigplan Notices**, 17(1), Proc. of CC'82, 1982.

[Rus92] David M. Russinoff. A verified Prolog Compiler for the Warren Abstract Machine. **Journal of Logic Programming**, 13:367–412, 1992.

[Sch86] D.A. Schmidt. **Denotational Semantics**. Allyn and Bacen, 1986.

[Sch95] D.A. Schmidt. Natural-Semantics-Based Abstract Interpretation (Preliminary Version). In **SAS'95**. 1995.

[Ses86] Peter Sestoft. The Structure of a self-applicable Partial Evaluator. In H. Ganzinger, N.D. Jones, editor, **Programs as Data Objects**, volume LNCS 217. Springer Verlag, 1986.

[Ses95] Peter Sestoft. Deriving a Lazy Abstract Machine. 1995.

[Set82] Ravi Sethi. Control Flow Aspects of Semantics Directed Compiling (Summary). **Sigplan Notices**, 17(1), Proc. of CC'82, 1982.

[SGJ94] M.H. Sørensen, R. Glück, and N. D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation and GPC. In D. Sannella, editor, **Programming Languages and Systems**, volume LNCS 788. Springer Verlag, 1994.

[SV84] U. Schmidt and R. Völler. A Multi-Language Compiler System with Automatically Generated Codegenerators. In **CC'84**. 1984.

[Tof90] Mads Tofte. **Compiler Generators - What they can do, what they might do, and what they will probably never do.**, volume 19. Springer, EATCS Monographs in Theoretical Computer Science, 1990.

[TS92] Christopher Tong and Duvvuru Sriram, editors. **Artificial Intelligence in Engineering Design**, volume I – III. Academic Press, 1992.

[Tur86] V.F. Turchin. The Concept of a Supercompiler. **ACM TOPLAS**, 8(3), 1986.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In **ESOP'88**, volume LNCS 300. Springer Verlag, 1988.

[Wan82] Mitchell Wand. Semantics-Directed Machine Architecture. In **Proc. of POPL'82**. 1982.

[Wan84] Mitchell Wand. A Semantic Prototyping System. In **CC'84**. 1984.

[Wan86] Mitchell Wand. From Interpreter to Compiler: A Representational Derivation. In H. Ganzinger, N.D. Jones, editor, **Programs as Data Objects**, volume LNCS 217. Springer Verlag, 1986.

[War77] David H.D. Warren. Implementing Prolog – compiling predicate logic programs. D.A.I Research Report No. 40, Edinburgh, 1977.

[Wei87] Pierre Weis. **Le Système SAM, Métacomputation très efficace à l'aide d'Operateurs sémantique**. PhD thesis, Université Paris VII, 1987.

[Win93] G. Winskel. **The Formal Semantics of Programming Languages**. MIT Press, 1993.

[WM92] Reinhard Wilhelm and Dieter Maurer. **Übersetzerbau – Theorie, Konstruktion, Generierung**. Springer Verlag, Berlin, Heidelberg, 1992. *English edition:* **Compiler Design***, Addison-Wesley, 1995.*

Index