Journal of Universal Computer Science, vol. 3, no. 5 (1997), 504-567
submitted: 20/12/96, accepted: 20/5/97, appeared: 28/5/97 © Springer Pub. Co.

On the Construction of Correct Compiler Back-Ends: An ASM Approach

Wolf Zimmermann
(University of Karlsruhe, Germany
zimmer@ipd.info.uni-karlsruhe.de)

Thilo Gaul
(University of Karlsruhe, Germany
gaul@ipd.info.uni-karlsruhe.de)

Abstract: Existing works on the construction of correct compilers have at least one of the following drawbacks: (i) correct compilers do not compile into machine code of existing processors. Instead they compile into programs of an abstract machine which ignores limitations and properties of real-life processors. (ii) the code generated by correct compilers is orders of magnitudes slower than the code generated by unverified compilers. (iii) the considered source language is much less complex than real-life programming languages. This paper focuses on the construction of correct compiler back-ends which generate machine-code for real-life processors from realistic intermediate languages. Our main results are the following: (i) We present a proof approach based on abstract state machines for bottom-up rewriting system specifications (BURS) for back-end generators. A significant part of this proof can be parametrized with the intermediate and machine language. (ii) The performance of the code constructed by our approach is in the same order of magnitude as the code generated by non-optimizing unverified C-compilers.

Key Words: Compiler, Operational Semantics, Verification, Abstract State Machine, Back-End Generator

Category: D.3.4, D.2.4

1 Introduction

Usually, correctness proofs of programs assume that programs are written in higher-level languages. However, any program is compiled into binary code and it is this code that is executed. Therefore, the correctness of programs depends also on the correctness of the compiler, and on the correctness of the processor. This paper discusses aspects for the construction of realistic correct compilers. Realistic correct compilers should produce machine code for real-life processors. The performance of the generated code should be comparable to machine code produced by usual compilers.

Any work on the construction of correct compilers must formalize the informal specification of the source and target languages, and assume that the implementation of the target language is correct. The correctness of compilers is then defined w.r.t. these formalizations. In our framework, we assume that the machine language of concrete processors and basic operating system routines (such as I/O, virtual address management) are implemented correctly. We consider imperative languages and concrete processors (here: the DEC-Alpha family). The semantics of imperative languages as well as machine languages of concrete processors is naturally described by state transformations. The formalization is based on *Abstract State Machines* (formerly: evolving algebras), because these are a well-suited device for formalizing state transformations. The correctness definition is based on *simulations of abstract state machines*. Our goal is to construct correct compilers that produce efficient code.

1.1 Related Work

In this subsection we analyze reasons why other methods to construct correct compilers fail to produce efficient machine code for real-life processors. The first work on correct compilers is [McCarthy and Painter 1967]. Most of the following work on correct compilation is based on denotational semantics (e.g. [Paulson 1981, Mosses 1982, Wand 1984, Brown et al. 1992, Mosses 1992, Palsberg 1992]), structural operational semantics (e.g. [Diehl 1996]), or on refinement e.g. [Buth et al. 1992], [Buth and Müller-Olm 1993], [Hoare et al. 1993], [Müller-Olm 1995], [Müller-Olm 1996], [Börger and Rosenzweig 1992], [Börger et al. 1994], [Börger and Durdanovic 1996]). Most of these works do not compile high-level programming languages into assembler languages. Instead, they design abstract machines, interpreters for these machines, and compile into code of these abstract machines. To our knowledge, only [Müller-Olm 1995, Müller-Olm 1996, Börger et al. 1994, Börger and Durdanovic 1996, Moore 1989] and ProCos [Hoare et al. 1993] discuss transformations into machine code.
[Börger et al. 1994, Börger and Durdanovic 1996] use also abstract state machines for the formalization of the source and target language.
The semantics-based approaches lead to monolithical compilers [Espinosa 1995, Tofte 1990]. Compilers constructed by these approaches translate into programs of machine-independent high-level abstract machines. These programs are interpreted. Consequently, the performance of the code generated by these compilers is poor and by three to four orders of magnitude slower than the code generated by compilers used in practice [Palsberg 1992]. From traditional compiler construction it is well-known that the introduction of intermediate languages is necessary for compiling programs of high-level languages into efficient machine code. Hence, our approach uses the concept of intermediate languages for the construction of the correct compilers. Additionally, this decomposition of the compilation simplifies our correctness proofs.
The refinement-based approaches preserve the program structure. Especially, the approaches refine expressions to a postfix form. Consequently, the code generated by these compilers is machine code for stack machines. Therefore, it is no surprise that all the works which consider compilation into machine code chose the transputer as the target machine [Müller-Olm 1995, Müller-Olm 1996, Börger et al. 1994, Börger and Durdanovic 1996]. However, it is well-known that the execution of a purely stack-based code is slow on register-based processors. In order to construct correct compilers which produce efficient machine code, it is necessary to reorganize the program structure. For the first time, this article considers correct compilation into machine-code of a register-based RISC-processor (the DEC-Alpha processor family).

1.2 Our Approach

There are two main issues that distinguish our approach for the construction of correct compilers from others: (i) we introduce intermediate languages, and (ii) the program structure is reorganized.
A compiler which compiles a higher level language SL into a machine language TL uses a sequence of intermediate languages $SL = IL_0, \ldots, IL_n = TL$, if the levels of languages differ too much. Instead of compiling SL-programs directly into TL-programs, SL-programs are compiled into IL_1-programs which are then compiled into IL_2-programs etc. The levels of the intermediate languages IL_i, IL_{i+1} do not differ too much. In this article, we will give a precise definition of this terminology. The concrete choice of intermediate languages is an engineering task. We therefore choose intermediate languages as used in the classical compiler architecture (see Figure 1). The intermediate languages differ from the source language that they are usually data structures representing the programs instead of being defined by a context-free grammar. Since our aim is to deal with compiler correctness, we formalize the notion of languages and their semantics such that source languages, intermediate languages and target languages are covered by this formalization.
The basic idea of a semantics definition is to define a family of abstract state machines, i.e. one abstract state machine per program. The reason for this decision is that it is convenient to distinguish the transformation of a program from mapping the state space. The state space may depend on the particular program.
We suggest the following method for proving correctness of compilations from an intermediate language IL_i to IL_{i+1}:

1. Merge the two languages
2. Prove the correctness of compilation by means of simulation of abstract state machines.

The former is based on homomorphisms between ASMs. The latter is similar to simulation proofs in complexity and computability theory.
One of the sources of inefficiencies in the generated code is the compilation of expressions. Therefore, we focus on this part of the compilation in order to demonstrate (ii). The compilation of expressions is a typical compiler back-end task. Compiler back-ends transform low-level intermediate language programs into machine programs. In this article we consider a typical class of intermediate languages which have the following characteristics: The program is represented as a set of basic block graphs. A basic block is a sequence of instructions where only the last instruction is a jump. Jump targets are restricted to the first instruction of basic blocks, i.e., each basic block has a unique label and these labels are used as operands of jumps.
Target machines usually have a different instruction set, a memory and some registers. The program is stored in the memory of the target machine. A program counter contains the address of the next instruction to be executed. Some or all of the arithmetic operations may use only other operands than registers or small integer or address values, i.e. expressions contained in intermediate language instructions must be implemented by a sequence of machine instructions.
Today, the components of compilers can be generated from a specification S of the transformation rules. One of the generation approaches for back-ends

assumes that S is given as a bottom-up term rewrite system [Emmelmann 1992, Proebsting 1995, Nymeyer et al. 1996, Nymeyer and Katoen 1996]. If we would have a correct generator applying the transformation rules, then it would be sufficient to prove the correctness of the compiling relation specified by S. We show that under general conditions, the correctness of S can be reduced to some local correctness conditions on single transformation rules, which can be proven independently. There are two simple proof strategies which check these local correctness conditions. These strategies are implemented in PVS. We proved a complete specification of a DEC-Alpha Back-End with these strategies using PVS. Therefore, the approach allows easy extensions of specifications.

In section 2, we introduce our basic terminology, define abstract state machines and homomorphisms on abstract state machines. Section 3 introduce our formalization of languages. In particular, it describes the framework for defining the data structures representing languages and the operational semantics. The latter is a template for defining a family of abstract state machines. Based on these definitions, the correctness of compilers and the notion of closely related languages is defined. Section 4 introduces our architecture for correct compilers, introduces term-rewriting system based construction of compiler back-ends and concludes with a precise definition of the problem solved in this article. The following sections show how correctness of compiler back-ends can be proven. Section 5 shows the decomposition of the problem into correctness requirements on the single term-rewrite rules. Section 6 shows how these requirements can be proven. Section 7 concludes our work. Appendix A defines the part of the abstract state machines for the example languages BB, BB_α, and L_α. We recommend to consult this appendix for the definition of requirements on languages. Appendix B shows how errors in the specifications of compiler back-ends can be found. It demonstrates this by an error which was detected during the application of our method. It would be hard to detect this error after an implementation of a compiler. Appendix C summarizes notations commonly used in this article.

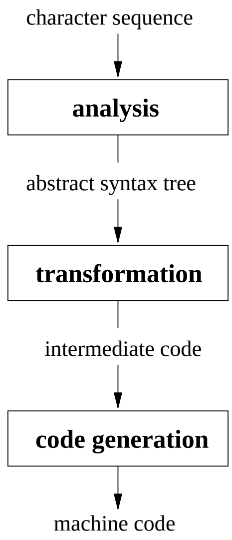

Figure 1: Architecture of Correct Compilers

2 Foundations

Our languages are defined operationally by abstract state machines (formerly evolving algebras) [Gurevich 1995]. Subsection 2.1 recalls the basic definitions and properties of signatures, algebras and term-rewrite systems used in this article. Subsection 2.2 defines the notation and properties of abstract state machine. The notation is taken from [Gurevich 1995].

2.1 Signatures, Algebras and Term Rewriting Systems

A *signature* Σ is a family of sets $(\Sigma_n)_{n\in\mathbb{N}}$. $f \in \Sigma_n$ is an *n-ary function symbol.* A *term over the signature* Σ (short:*Σ-term*) is defined as usual. $T(\Sigma)$ denotes the set of terms over the signature Σ. A *Σ-algebra* $\mathcal{A}$ consists of a carrier set U and of a total function $[\![f]\!]_\mathcal{A} : U^n \to U$ for each $f \in \Sigma_n$, $n \in \mathbb{N}$. $[\![f]\!]_\mathcal{A}$ is the *interpretation of f in algebra $\mathcal{A}$*. $[\![\cdot]\!]_\mathcal{A}$ can be extended to $T(\Sigma)$ by defining $[\![f(t_1,\dots,t_n)]\!]_\mathcal{A} = [\![f]\!]_\mathcal{A}([\![t_1]\!]_\mathcal{A},\dots,[\![t_n]\!]_\mathcal{A})$ for each n-ary function $f \in \Sigma_n$, $t_i \in T(\Sigma)$, $i = 1,\dots,n$. We omit the index $\mathcal{A}$ if it is clear from the context.
Throughout the article, we assume that there is an element $\perp \in \Sigma_0$ representing undefined values. A *universe* V is a predicate such that $[\![V]\!]_\mathcal{A}$ is identified with the set $\{x : [\![V]\!]_\mathcal{A}(x)\}$ for any Σ-algebra $\mathcal{A}$. The set of universes $S \subseteq \Sigma$ is called the *sorts* of Σ. The universe $BOOL$ is defined by $[\![BOOL]\!] = \{\mathit{true}, \mathit{false}\}$. A n-ary function $f : U_1 \times \cdots U_n \to V \in \Sigma_n$ from universe $U_1 \times \cdots \times U_n$ to an universe V is an n-ary operation on the carrier set U such that $[\![f]\!](a_1,\dots,a_n) \in V$ for all $a_1 \in U_1, \dots a_n \in U_n$ and $[\![f]\!](a) = \perp$ otherwise. A term $t \in T(\Sigma)$ is a *U-term* iff $[\![t]\!] \in [\![U]\!]$, denoted by $t \in U$. An algebra with carrier set $T(\Sigma)$,

$$[\![f]\!](t_1,\dots,t_n) = \begin{cases} f(t_1,\dots,t_n) & \text{if } t_1 \in U_1,\dots,t_n \in U_n \\ \perp & \text{otherwise} \end{cases}$$

for each $f : U_1 \times \cdots \times U_n \to V \in \Sigma_n \setminus S$, and $[\![V]\!] = \{f(t_1,\dots,t_n) : f : U_1 \times \cdots \times U_n \to V \wedge t_1 \in U_1 \cdots t_n \in U_n\}$ is the *Σ-term algebra*. We assume that each $f \in \Sigma$ is strict in $\perp$, i.e.$[\![f]\!](\cdots\perp\cdots) = \perp$. $U \sqsubseteq V$ denotes the fact that $[\![U]\!]_{\mathcal{T}(\Sigma)} \subseteq [\![V]\!]_{\mathcal{T}(\Sigma)}$ for sorts U, V. Since $\mathcal{T}(\Sigma)$ is initial among the Σ-algebras, $[\![U]\!]_\mathcal{A} \subseteq [\![V]\!]_\mathcal{A}$ for all Σ-algebras $\mathcal{A}$.
In the following, we use the data-types in table 1 without further explanation. $x.l$ is an abbreviation for $\mathit{cons}(x,l)$ and $\circ$ denotes the concatenation of lists. $\mathit{snoc}(l,x)$ adds element x to the end of list l. l_i denotes the i-th element of a list l, and $l\langle i:j\rangle$ denotes the sublist $\langle l_i, l_{i+1},\dots,l_j\rangle$. The type $\mathbb{N}$ denotes the universe of natural numbers. We use the usual arithmetic operations. The concrete use

Type	Meaning	Operations
$A \times B$	pairs consisting of type A and B	$(\cdot,\cdot)$, *fst*, *snd*
T^*	list of elements of type T	*cons*, $\langle\rangle$, *tail*, *front*, l_i
$BOOL$	truth values	the logical operators

Table 1: Standard Data-Types

becomes clear from the context.
Let Σ and Σ' be two signatures with sorts S and S', respectively. A *signature morphism* maps the sorts and function symbols of the one signature on the sorts and function symbols of the other signature, i.e. it is a mapping $\sigma : \Sigma \to \Sigma'$such that $\sigma(f) : \sigma(U_1)\times\cdots\times\sigma(U_n) \to \sigma(V) \in \Sigma'$ for every $f : U_1\times\cdots\times U_n \to V \in \Sigma$. Mappings $\phi : T(\Sigma) \to T(\Sigma')$ may be defined by a basis, i.e. a mapping $\bar\phi : \Sigma \to \Sigma' \cup T(\Sigma')$ such that for every $f \in \Sigma_i$, $i > 0$, $\bar\phi(f) \in \Sigma'$. The mapping $\phi : T(\Sigma) \to T(\Sigma')$ defined by $\phi(f(t_1,\dots,t_n)) = \bar\phi(f)(\phi(t_1,\dots,t_n))$ is the *canonical*

extension of $\bar{\phi}$. Any mapping $\phi : S \to T(\Sigma')$ where S is a finite set of Σ-terms can be extended canonically in a similar way. In this case we use also ϕ to denote this canonical extension.

Let $\mathcal{A}$ be a Σ-algebra and $\mathcal{A}'$ be a Σ'-algebra with carrier sets U and U', respectively. A mapping ψ from $\mathcal{A}$ into $\mathcal{A}'$ must map operators of $\mathcal{A}$ to operators of $\mathcal{A}'$ and the universe U to the universe U'. Suppose $\phi : T(\Sigma) \to T(\Sigma')$ is a canonical extension of a mapping $\bar{\phi} : \Sigma \to \Sigma' \cup T(\Sigma')$. It is useful if ψ is compatible to ϕ, i.e. $\psi(\llbracket\perp\rrbracket_{\mathcal{A}}) = \llbracket\perp\rrbracket_{\mathcal{A}'}$, $\psi(\llbracket f\rrbracket_{\mathcal{A}}) = \llbracket\bar{\phi}(f)\rrbracket_{\mathcal{A}'}$ for all $f \in \Sigma_0$ and $\psi(\llbracket f\rrbracket_{\mathcal{A}}(a_1,\ldots,a_n)) = \llbracket\bar{\phi}(f)\rrbracket_{\mathcal{A}'}(\psi(a_1),\ldots,\psi(a_n))$ for all $a_1,\ldots,a_n \in U$, $f \in \Sigma \setminus \Sigma_0$. Observe that ϕ maps terms to terms while ψ maps interpretations of terms to interpretations of terms. Such a mapping $\psi : \mathcal{A} \to \mathcal{A}'$ is called a *$\bar{\phi}$-algebra homomorphism.* It is not hard to prove that $\psi(\llbracket t\rrbracket_{\mathcal{A}}) = \llbracket\phi(t)\rrbracket_{\mathcal{A}'}$ for all $t \in T(\Sigma)$. A $\bar{\phi}$-homomorphism $\psi : \mathcal{A} \to \mathcal{A}'$ is a *mono-morphism* (*epi-morphism*, *isomorphism*) iff ψ' is injective (surjective, bijective).

Let Σ be a signature with sorts S. A *Σ-algebra homomorphism* $\phi : \mathcal{A} \to \mathcal{A}'$ is a mapping $\phi' : U \to U'$ such that $\phi'(\perp) = \perp$ and $\phi'(\llbracket f\rrbracket_{\mathcal{A}}(a_1,\ldots,a_n)) = \llbracket f\rrbracket_{\mathcal{A}'}(\phi'(a_1),\ldots,\phi'(a_n))$ for each $n \in \mathbb{N}$, $f \in \Sigma_n$ and each $a_i \in U$. Observe that $a \in \llbracket V\rrbracket_{\mathcal{A}}$ implies $\phi'(a) \in \llbracket V\rrbracket_{\mathcal{A}'}$ for every universe $V \in \Sigma$.

Let Σ and Σ' be two signatures such that $\Sigma \subseteq \Sigma'$ and $\mathcal{A}'$ be a Σ'-algebra with carrier set U'. We can restrict $\mathcal{A}'$ to the interpretation of $f \in \Sigma$ and Σ-terms. The *σ-restriction* $\mathcal{A}'|_{\Sigma}$ is the algebra with the carrier set U' and the the operations $\llbracket f\rrbracket_{\mathcal{A}'}$ for each $f \in \Sigma$.

Throughout the article we use the following:

Assumption: Any mapping $\bar{\phi} : \Sigma \to T(\Sigma')$,and $\bar{\phi}$-algebra homomorphism ϕ preserve $BOOL$, the logical constants *true* and *false*, and the logical operators. ■

Let Σ be a signature and V be a set of symbols disjoint from Σ. $T(\Sigma, V)$ denotes the set of terms over signature Σ and variables V. A *substitution* $\sigma : T(\Sigma,V) \to T(\Sigma,V)$ is the canonical extension of a mapping $\bar{\sigma} : V \to T(\Sigma,V)$ where $\bar{\sigma}(v) \neq v$ for only a finite number of variables. We denote substitutions by $\sigma = [x_1/t_1]\ldots[x_n/t_n]$ where $\sigma(x_i) = t_1$ and $\sigma(v) = v$ for $v \neq x_i$. Subterms of a term a denoted by occurences. An *occurence* is a finite list of natural numbers. The *subterm of term t at occurence o*, denoted by $t[o]$ is recursively defined as follows: $t[\langle\rangle] = t$ and $t[snoc(o,i)] = t_i$ if $t[o] = f(t_0,\ldots,t_{n-1})$ for a $f \in \Sigma_n$. In this case $t[snoc(o,i)]$ is undefined if $i \geq n$. $t[o/u]$ denotes the term t where the subterm at o is replaced by term u. A term $t \in T(\Sigma,V)$ matches a term $t' \in T(\Sigma,V)$ iff there is a substitution such that $\sigma(t) = t'$.

A *term-rewrite rule over signature Σ and variables V* is a pair $t \hat{=} t'$ of terms $t, t' \in T(\Sigma,V)$ where each variable occurring in t' also occurs in t. A *term-rewrite system* (TRS) is a set of term-rewrite rules. Let R be a TRS. A term $t \in T(\Sigma)$ *rewrites into* a term $t' \in T(\Sigma)$, denoted by $t \hookrightarrow_R t'$, iff there is a rule $lhs \hat{=} rhs \in R$ and an occurence o such that $\sigma(lhs) = t[o]$ and $t' = t[o/\sigma(rhs)]$.

As usual, $\overset{+}{\hookrightarrow}_R$ denotes the transitive closure and $\overset{*}{\hookrightarrow}_R$ the reflexive, transitive closure of $\hookrightarrow_R$. The notion of *normal forms*, *noetherian* and *confluent* TRS is defined as usual. $NF_R(t)$ denotes the (unique) normal form of term t for noetherian and confluent TRS.

A conditional term-rewrite rule is a quadruple of terms with variables, denoted by **if** $t_1 = t_2$ *mathbfthen* $t_3 \hat{=} t_4$. A conditional TRS is a set of conditional or unconditional term-rewrite rules. Let $\mathcal{A}$ be a Σ-algebra. A term $t \in T(\Sigma)$ $\mathcal{A}$-

rewrites into a term $t' \in T(\Sigma)$, iff either there is an unconditional rule $lhs \hat{=} rhs$ such that there is an occurence o such that $\sigma(lhs) = t[o]$ and $t' = t[o/\sigma(rhs)]$, or there is a conditional rule **if** $t_1 = t_2$ **then** $lhs \hat{=} rhs$ such that $\sigma(lhs) = t[o]$ and $t' = t[o/\sigma(rhs)]$, and $[\![\sigma(t_1)]\!]_{\mathcal{A}} = [\![\sigma(t_2)]\!]_{\mathcal{A}}$. The above definitions can be extended straightforwardly to BURS and conditional TRS.
$\bar{\phi}$-homomorphisms carry over to TRS:

Theorem 1. Let Σ and Σ' be two signatures, R be a noetherian and confluent TRS, $\bar{\phi} : \Sigma \to \Sigma' \cup T(\Sigma')$ be a mapping, and $\phi : T(\Sigma, V) \to T(\Sigma', V)$ be its canonical extension. Then, the following properties hold:

(a) Let $\sigma : V \to T(\Sigma)$ be a substitution and $\sigma' : V \to T(\Sigma')$ be the substitution such that $\sigma'(v) = \phi(\sigma(v))$ for all $v \in V$. Then, $\sigma'(\phi(t)) = \phi(\sigma(t))$ for all $t \in T(\Sigma)$.

(b) Let $\phi(R) = \{\phi(lhs) \hat{=} \phi(rhs) : lhs \hat{=} rhs \in R\}$. Then $t \overset{*}{\hookrightarrow}_R t'$ implies $\phi(t) \overset{*}{\hookrightarrow}_R \phi(t')$. If $\phi(t)$ is a normal form, then t is a normal form. If $\phi(R)$ is confluent then R is confluent. If $\phi(R)$ is noetherian, then R is noetherian. These implications are equivalences, if ϕ is injective. If $\phi(R)$ is noetherian and confluent, then $NF_{\phi(R)}(\phi(t)) = \phi(NF_R(t))$ for all $t \in T(\Sigma)$. ■.

2.2 Abstract State Machines

In this subsection we introduce the notion of ASMs and ASM-homomorphisms. An *abstract state machine* (short: ASM) is a tuple $\mathcal{A} = (\Sigma, Q, S, \to, I)$, where Σ is a signature, Q is a set of Σ-algebras (the *states*) with the same carrier set, S is a set of sorts (the *super-universe*), $\to \subset Q \times Q$ is the *transition relation*, and $I \subset Q$ is the set of *initial states*. The relation $\to$ is defined by a finite collection of transition rules of the form

if *Cond* **then** *Updates* **endif**.

where $Cond \in BOOL$ and *Update* is a finite set of updates, i.e. of pairs $lhs := rhs$, $lhs, rhs \in T(\Sigma)$. A rule is *applicable in state* q' iff $[\![Cond]\!]_{q'} = true$. Let q' be a state before and q be a state after applying an applicable rule. Then, for any update $f(t_1, \ldots, t_n) := t$, we have

$$[\![f]\!]_q(x_1, \ldots, x_n) = \begin{cases} [\![t]\!]_{q'} & \text{if for all } i,\ 1 \le i \le n,\ [\![t_i]\!]_{q'} = x_i \\ [\![f]\!]_{q'}(x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

If $[\![f(t_1, \ldots, t_n)]\!]_{q'} \neq [\![t]\!]_{q'}$, we also say that $q' \to q$ *executes the update* $f(t_1, \ldots, t_n) := t$. If several rules are applicable, then one applicable rule is chosen nondeterministically. As usual, $\overset{n}{\to}$ denotes the composition of n state transitions, where the composition of relations is defined as usual, i.e. $\rho_1 \circ \rho_2 = \{(u, w) : \exists v : (u, v) \in \rho_1 \wedge (v, w) \in \rho_2\}$. $\overset{*}{\to}$ denotes the reflexive, transitive closure and $\overset{+}{\to}$ denotes the transitive closure of $\to$. A state $q \in Q$ is *reachable* iff there is an initial state $i \in I$ such that $i \overset{*}{\to} q$. W.l.o.g. we assume that each $q \in A$ is reachable. The set $F = \{f \in Q : \forall q' \in Q : f \not\to q'\}$ is the set of *final states*. An ASM is *deterministic* iff for each $q \in Q$ there is at most one $q' \in Q$ such that $q \to q'$.
Notations: Let *rule₁* and *rule₂* be transition rules or sets of updates.
if *Cond* **then** $rule_1$ **else** $rule_2$ is an abbreviation for the two transition rules
if *Cond* **then** $rule_1$ and **if** $\neg Cond$ **then** $rule_2$.

if *cond₁* **then**
  if *cond₂* **then** *rule₁*
  else *rule₂*

is an abbreviation for the two transition rules **if** $cond_1 \wedge cond_2$ **then** $rule_1$ and **if** $cond_1 \wedge \neg cond_2$ **then** $rule_2$. If the **else**-branch is omitted then the latter transition rule is omitted. ■

We distinguish the following classes of functions: *Dynamic functions*: the interpretation of a dynamic function is changed by transition rules, i.e. f is called a dynamic function if an assignment of the form $f(t_1, \ldots, t_n) := t_{n+1}$ appears in a transition rule. *Static functions*: the interpretation of a static function is never changed.

Let $\Xi \subseteq \Sigma$ be the set of static functions of an ASM $\mathcal{A} = (\Sigma, Q, S, \rightarrow, I)$. The restrictions $q|_\Xi$ to Ξ and $q'|_\Xi$ to Ξ are identical for all $q, q' \in Q$. The Ξ-algebra $q|_\Xi$ is the *static algebra* of $\mathcal{A}$. Universes, *true*, *false* and $\perp$ are always static functions. We assume that the elements of each sort are representable by static functions, i.e. for each sort $V \in S$, $x \in [\![V]\!]_\mathcal{X}$, there is a Ξ-term t such that $[\![t]\!]_\mathcal{X} = x$. This implies that for each state q and Σ-term t, there is a Ξ-term t' such that $[\![t]\!]_q = [\![t']\!]_{q'}$. However, the term t' may be different for different states. This property allows to discuss state changes on the basis of Ξ-terms.

External functions allow interaction with the outside world. They need not to be specified, only some requirements may be specified. Any interpretation of this function satisfies at least these requirements. External functions are never changed explicitly by a transition rule. However an external function may have different interpretations in different states.

Let $\mathcal{A}_1 = (\Sigma_1, Q_1, S_1, \rightarrow_1, I_1)$, $\mathcal{A}_2 = (\Sigma_1, Q_2, S_2, \rightarrow_2, I_2)$ be two ASMs with static parts $\mathcal{X}_1$ and $\mathcal{X}_2$, respectively. An *ASM-homomorphism* $\xi : \mathcal{A}_1 \rightarrow \mathcal{A}_2$ recovers the ASM $\mathcal{A}_1$ within $\mathcal{A}_2$. Formally, ξ is a triple $(\bar{\phi}, \psi, \gamma)$ consisting of a mapping $\bar{\phi} : \Sigma_1 \rightarrow \Sigma_2 \cup T(\Sigma_2)$, a $\bar{\phi}$-homomorphism $\psi : \mathcal{X}_1 \rightarrow \mathcal{X}_2$, and a mapping $\gamma : Q_1 \rightarrow Q_2$ such that the following conditions are satisfied for all $q, q' \in Q_1$:

(H1) For all $t \in T(\Sigma_1)$ is $\psi([\![t]\!]_q) = [\![\phi(t)]\!]_{\gamma(q)}$, where ϕ is the canonical extension of $\bar{\phi}$.
(H2) $\gamma(I_1) \subseteq I_2$
(H3) $q \rightarrow_1 q'$ implies $\gamma(q) \rightarrow_2 \gamma(q')$.

An ASM-homomorphism ξ is a *monomorphism* (*epimorphism*, *isomorphism*), iff ϕ is injective (surjective,bijective), ψ is a monomorphism (epimorphism, isomorphism) and γ is injective (surjective bijective). The updates and transition rules can be mapped by ϕ in the straighforward way, i.e. $\phi(\mathit{lhs} := \mathit{rhs}) = \phi(\mathit{lhs}) := \phi(\mathit{rhs})$ and

$$\phi(\textbf{if } \mathit{cond} \textbf{ then } \mathit{Updates}) = \textbf{if } \phi(\mathit{cond}) \textbf{ then } \phi(\mathit{Updates}).$$

We generalize the notion of a canonical extension by these definitions.

Lemma 1 Let $\xi = (\bar{\phi}, \psi, \gamma) : \mathcal{A}_1 \rightarrow \mathcal{A}_2$ be an ASM-homomorphism and ϕ be the canonical extension of $\bar{\phi}$. If $\gamma(q') \rightarrow_2 \gamma(q)$ executes the update $\phi(\mathit{lhs}) := \phi(\mathit{rhs})$ then $q' \rightarrow_1 q$ executes the update $\mathit{lhs} := \mathit{rhs}$. If ξ is a mono-morphism, then the converse is also true.

Proof: Suppose the contrary, i.e. $\gamma(q') \to_2 \gamma(q)$ executes the update $\phi(lhs) := \phi(rhs)$ but $q' \to_1 q$ does not execute the update $lhs := rhs$. Suppose $lhs = f(t_1, \ldots, t_n)$. Then there are $t_i' \in T(\Xi_1)$ such that $[\![lhs]\!]_{q'} = [\![f(t_1', \ldots, t_n')]\!]_{q'}$. Let lhs' denote this term. Thus (1) $[\![lhs']\!]_q = [\![lhs']\!]_{q'}$, (2) $[\![\phi(lhs')]\!]_{\gamma(q)} = [\![\phi(rhs)]\!]_{\gamma(q')}$, and (3) $[\![\phi(lhs')]\!]_{\gamma(q')} \neq [\![\phi(rhs)]\!]_{\gamma(q')}$. Obviously, (1) and (H1) imply that $[\![\phi(lhs')]\!]_{\gamma(q')} = [\![\phi(lhs')]\!]_{\gamma(q)}$. This implies together with (2) that $[\![\phi(lhs')]\!]_{\gamma(q')} = [\![\phi(rhs)]\!]_{\gamma(q')}$ in contradiction to (3). If ξ is a mono-morphism, the assumption that the converse is violated leads to a similar contradiction using (H1) and the fact that ψ is a mono-morphism. ■

Lemma 1 can be used to define γ inductively, based on an injective mapping $\gamma : I_1 \to I_2$ satisfying (H1) for all $i \in I_1$.

Lemma 2 (Defining γ) Let $\mathcal{A}_1 = (\Sigma_1, Q_1, S_1, \to_1, I_1)$, $\mathcal{A}_2 = (\Sigma_1, Q_2, S_2, \to_2, I_2)$ be two ASMs with static parts $\mathcal{X}_1$ and $\mathcal{X}_2$, respectively. Furthermore, let $\bar{\phi} : \Sigma_1 \to \Sigma_2 \cup T(\Sigma_2)$ be an injective mapping, $\psi : \mathcal{X}_1 \to \mathcal{X}_2$ be a $\bar{\phi}$-mono-morphism, and $\gamma : I_1 \to I_2$ be an injective mapping satisfying (H1) for all $i \in I_1$. Then γ can be extended to an injective mapping $\gamma : Q_1 \to Q_2$ such that $\xi = (\bar{\phi}, \psi, \gamma) : \mathcal{A}_1 \to \mathcal{A}_2$ is an ASM-homomorphism.

Proof: Let $q \in Q_1$, we extend γ such that $i \overset{n}{\to}_1 q$ implies $i \overset{n}{\to}_2 \gamma(q)$. γ is defined by induction on n. The base case $n = 0$ is obvious. If $n > 0$, then there is a state q' such that $i \overset{n-1}{\to}_1 q' \to_1 q$. By induction, we have γ already extended such that $\gamma(i) \overset{n-1}{\to}_2 \gamma(q')$. Let **if** *cond* **then** *Updates* the transition rule applied on q' such that $q' \to_1 q$. By the induction hypothesis, $[\![\phi(cond)]\!]_{\gamma(q')} = \psi([\![cond]\!]_q) = true$. Thus, there is a state $\bar{q} \in Q_2$ obtained from $\gamma(q')$ by the updates $\phi(Updates)$, i.e. $\gamma(q') \to \bar{q}$. Defining $\gamma(q) = \bar{q}$ will do the job: (H3) is obviously satisfied. (H1) can be shown by an easy structural induction on t. ■

Thus, it is sufficient to define γ on the initial states.

3 Languages

In this section, we formalize programming languages by ASMs. Our formalization captures the structure of programs (Subsection 3.1) as well as their operational semantics (Subsection 3.2). Since the formalization is used for proving the correctness of compilers or constructing correct compilers, it is convenient to define all languages used in a compiler (source language, target language, intermediate languages) within the same framework. The natural view on target machines and imperative programming languages is an operational view, i.e. instructions are executed which transform states. Consequently, it is natural to define the semantics operational by ASMs. The concrete examples of languages considered in this article are discussed in appendix A. The reader may consult this appendix to understand our motivations.

Every program interacts with its environment (e.g. I/O). Thus, parts of a state can be *observed* by an environment (e.g. read from an input stream or write into an output stream). The *observable behavior* are the state changes of these parts. Informally, a correct compiler needs only to ensure the preservation of the observable behavior. Subsection 3.3 shows this formalization of the notion of correct compilers. It is based on a notion of simulation of ASMs similar to

complexity and computability. We show decomposition theorems which allow the introduction of intermediate languages (vertical decomposition) and intermediate states in computations (horizontal decomposition).
The basic idea of constructing a correct compiler is to design a sequence of intermediate languages $IL_1, \ldots, IL_n$ such that the languages IL_i and IL_{i+1} are closely related. This relation is formalized in subsection 3.4.

3.1 Structure of Programming Languages

The formalization of the notion of a language should capture all languages used by a compiler. After syntax analysis, source programs are usually trees; intermediate programs are some data structures which may contain control flow information (e.g. basic block graphs), and target programs are sequences of words stored in the memory of the target machine. As discussed above, we assume that any language has the notion of an instruction. For the operational semantics, it is convenient to assume that programs define a control flow, i.e. an execution order on instructions.
A *language* is a tuple $L = (\Sigma_L, S_L, \Gamma_L, INSTR, PROG, \mathit{well\_defined}_L, \mathcal{I}_L)$ where Σ_L is a signature (the *program structure*), S_L is a set of sorts, Γ_L is a signature (the *control flow*), $INSTR \in S_L$ is the sort of *instructions*,$PROG \in S_L$ is the sort of *programs*, $\mathit{well\_defined}_L : PROG \to BOOL$ is a predicate (the *static semantics*), and $\mathcal{I}_L$ is a $(\Sigma_L \cup \Gamma_L \cup \{\mathit{well\_defined}_L\})$-algebra where $\mathcal{I}_L|_{\Sigma_L} = \mathcal{T}(\Sigma_L)$, i.e. the restriction of $\mathcal{I}_L$ to Σ_L is equal to Σ_L-term algebra. The *signature of instructions* is the largest set $\Upsilon_L \subseteq \Sigma_L$ satisfying (i) $INSTR \in \Upsilon_L$ for every $f : T_1 \times \cdots \times T_k \to INSTR \in \Upsilon_L$ or (ii) for every $f : T_1 \times \cdots \times T_k \to T \in \Sigma$, $f : T_1 \times \cdots \times T_k \to T \in \Upsilon_L$ and $T_1, \ldots, T_k \in \Upsilon_L$, if $T \in \Upsilon_L$, i.e., Υ_L contains the constructor for building instructions.
$INSTR$-terms correspond to instructions and $PROG$-terms correspond to programs. Obviously, $T(\Upsilon)$ contains all $INSTR$-terms. The functions in Σ_L define the abstract syntax tree of programming languages. The sorts in S_L represent syntactic constructs. $\mathit{well\_defined}_L$ defines some (static) semantic conditions on programs (e.g. correct typings etc.). $\pi \in L$ denotes that π is a well-defined $PROG$-term, i.e. $[\![\mathit{well\_defined}(\pi)]\!]_{\mathcal{I}_L} = \mathit{true}$. A language L_1 is a *sublanguage* of L_2, denoted by $L_1 \subseteq L_2$ iff $\pi \in L_1$ implies $\pi \in L_2$.
Remark: We do not consider the styles how languages can be defined. For higher imperative programming languages, e.g. it is possible to define Σ_L by context-free grammars. The tree representation of a structure tree corresponds then uniquely to an abstract syntax tree. The whole language may be defined by *Montages*[Pierantonio and Kutter 1997] since they also define control flow, static semantics and instructions. ■

Example 1 (Basic Block Graphs) Figure 2 shows an example program of the language *BB* of basic block graphs, defined in (cf. Appendix A.1). The blocks are given by boxes, labels are numbers. ■

Notation: Let L be a language and $U \in S_L$. U_π denotes the set of U-terms which are sub-terms of π. In particular $INSTR_\pi$ denotes the set of instructions of program π. E.g. consider the program in Figure 2. Then we have $LABEL_\pi =$

0:	$readint(local(intconst_1))$ $condjump(intequal(cont(local(intconst_1)), intconst_0), 0, 1)$
1:	$stop$

Figure 2: A Basic Block Program

$\{0, 1\}$,

$$EXPR_\pi = \left\{ \begin{array}{c} intconst_0, intconst_1, local(intconst_1) \\ intequal(local(intconst_1), intconst_0), 0, 1) \end{array} \right\},$$

$$INSTR_\pi = \left\{ \begin{array}{c} readint(local(intconst_1)) \\ condjump(intequal(local(intconst_1), intconst_0), 0, 1) \\ stop \end{array} \right\},$$

and $BLOCK_\pi$ is the set of the two blocks depicted in Figure 2. ■
Appendix A contains the parts of the definitions of the example languages used in this article.

3.2 Operational Semantics of Programming Languages

In our approach, each program $\pi \in L$ has its own ASM $\mathcal{A}_\pi$. The basic idea is that $\mathcal{A}_\pi$ has an instruction pointer IP referring to an instruction of π, and the transition rule for the instruction referred by IP defines the updates. However, the signature of the dynamic functions and static functions of these ASMs are identical, the transition rules for particular instructions $f(t_1, \ldots, t_n)$ for the same functor f have the same shape, and the initial states are closely related. The components where the ASMs differ are the universes and the state space. E.g., the interpretation of IP is always an $INSTR_\pi$-term. At the end of this section we formalize the notion of observable behavior of programs, i.e. the behavior which can be observed by the environment on which the programs interact. The basic idea for defining an operational semantics is that an operational semantics serves as a template such that $\mathcal{A}_\pi$ can be derived from this template.
In the following let L be a language. A *static part of an operational semantics* L is a triple $Stat_L = (\Xi, U, \mathcal{X})$ where Ξ is a signature satisfying $\Sigma_L \cup \Gamma_L \subseteq \Xi$, U is a set of sorts satisfying $S_L \subseteq U$, and $\mathcal{X}$ is a Ξ-algebra such that $\mathcal{X}|_{\Sigma_L \cup \Gamma_L} = \mathcal{I}_L$. The static part of a language L is used to model the static algebra of the ASMs for $\pi \in L$. $\Psi_L = \Xi_L(\Sigma_L \cup \Gamma_L)$ denotes the static functions not used for defining programs.
A *$Stat_L$-signature of the dynamic part of an operational semantics of L* is a pair $Dyn_L = (\Delta, IP, \Omega)$ where Δ is a signature (*dynamic functions*) satisfying $\Delta \cap \Xi = \emptyset$, $IP \in T(\Delta \cup \Xi)$ is an $INSTR$-term (the *instruction pointer*), and $\Omega \subseteq \Delta$ is the set of *observable functions*.
Notation: We use the constant *prog* which is substituted by the program $\pi \in L$ for the definition of the ASM $\mathcal{A}_\pi$ for π. ■.
Δ are the dynamic functions of the ASMs for $\pi \in L$. The observable functions are those dynamic functions (constants) which can be observed by the environment.

Example 2 (Basic Block Graphs, continued) (cf. Appendix A.1) The static part of the operational semantics of BB contains the operations on the data types of BB. These data types and operations are the same as on the DEC-Alpha. The instruction pointer is implicitly given by the block pointer BP and the program counter PC referring to the instructions within a block. Other dynamic functions model the memory (function *content*), pointers to the local and global environment, and I/O-streams. The latter are observable. ■

An operational semantics must contain information to define the initial states and the transition relation of the ASMs for $\pi \in L$. The idea is to define initial states by updates and transition rules by instantiating some rules with variables and expanding macros.
In the following definitions let $Stat_L$ be a static part of an operational semantics L, Dyn_L a $Stat_L$-signature of a dynamic part of an operational semantics of L, Θ be a signature satisfying $\Theta \cap (\Xi \cup \Delta) = \emptyset$, and V a set of variables where $V \cap (\Theta \cup \Xi \cup \Delta) = \emptyset$. Θ is used later for the signature of macros.
A $(Stat_L, Dyn_L, \Theta, V)$-*macro* is a term-rewrite rule $lhs \hat{=} rhs$ where $lhs, rhs \in T(\Theta \cup \Xi \cup \Delta, V)$. A $(Stat_L, Dyn_L, \Theta, V)$-*update* is a pair $lhs := rhs$ where $lhs = f(t_1, \ldots, t_n)$ for a $f \in \Delta$, $t_1, \ldots, t_n \in T(\Theta \cup \Xi \cup \Delta, V)$ and $rhs \in T(\Theta \cup \Xi \cup \Delta, V)$. Updates are refined into updates performed by the ASMs for the programs $\pi \in L$. A $(Stat_L, Dyn_L, \Theta, V)$-*rule* is pair $IP = f(x_1, \ldots, x_k) \rightsquigarrow rhs$ where $x_1, \ldots, x_k \in V$,

(O1) $f : T_1 \times \cdots \times T_K \to INSTR$ for $T_1, \ldots, T_k \in \Theta_L$, and
(O2) there is an $m \geq 0$ such that[1]
$$\begin{aligned} rhs = \ & \textbf{if } cond_1 \textbf{ then } Updates_1 \\ & \textbf{elsif } cond_2 \textbf{ then } Updates_2 \\ & \vdots \\ & \textbf{elsif } cond_m \textbf{ then } Updates_m \\ & \textbf{else } Updates_0, \end{aligned}$$
$cond_1, \ldots, cond_m \in T(\Theta \cup \Xi \cup \Delta, V)$, and $Updates_0, \ldots, Updates_m$ are sets of $(Stat_L, Dyn_L, \Theta, V)$-updates.

$f(x_1, \ldots, x_k) \rightsquigarrow rhs$ is *closed* if rhs contains at most the variables $\{x_1, \ldots, x_k, \pi\}$. Closed $(Stat_L, Dyn_L, \Theta, V)$-rules are refined into transition rules of the ASMs for the programs $\pi \in L$ by substituting the variables. A substitution σ can be extended to updates and rules straightforwardly.
Notation: We use the following conventions to denote $(Stat_L, Dyn_L, \Theta, V)$-rules. A rule $f(x_1, \ldots, x_k) \rightsquigarrow rhs$ is denoted by

$$\textbf{if } IP = f(x_1, \ldots, x_k) \textbf{ then } Updates$$

If $Updates_0 = \emptyset$ then the **else**-branch is omitted.

$$\begin{aligned} & \textbf{if } IP = f(x_1, \ldots, x_{i-1}, t, x_{i+1}, \ldots, x_k) \\ & \textbf{then if } cond_1 \textbf{ then } Updates_1 \\ & \quad \textbf{elsif } cond_2 \textbf{ then } Updates_2 \\ & \quad \vdots \\ & \quad \textbf{elsif } cond_m \textbf{ then } Updates_m \\ & \quad \textbf{else } Updates_0 \end{aligned}$$

[1] $m = 0$ means $rhs = Updates_0$.

for a term $t \in T(\Sigma_L)$ is an abbreviation for

$$
\begin{array}{l}
\mathbf{if}\ IP = f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_k) \\
\mathbf{then\ if}\ x_i = t \wedge cond_1\ \mathbf{then}\ Updates_1 \\
\qquad \mathbf{elsif}\ x_i = t \wedge cond_2\ \mathbf{then}\ Updates_2 \\
\qquad \vdots \\
\qquad \mathbf{elsif}\ x_i = t \wedge cond_m\ \mathbf{then}\ Updates_m \\
\qquad \mathbf{elsif}\ x_i = t\ \mathbf{then}\ Updates_0\ \blacksquare
\end{array}
$$

An *operational semantics of* L is a tuple

$$\mathbb{A}_L = (Stat_L, Dyn_L, M, V, Macros, Init, Trans)$$

where $Stat_L$ is the static part of $\mathbb{A}_L$, Dyn_L is the $Stat_L$-signature of the dynamic part of $\mathbb{A}_L$, Θ is a signature satisfying $\Theta \cap (\Xi \cup \Delta) = \emptyset$ (the signature of *macros*), V a set of *variables* satisfying $V \cap (\Theta \cup \Xi \cup \Delta) = \emptyset$,

(O3) *Macros* is a set $(Stat_L, Dyn_L, \Theta, V)$-macros defining a noetherian and confluent TRS such that $NF(t) \in T(\Xi \cup \Delta)$ for every $t \in T(\Xi \cup \Delta \cup M)$,
(O4) *Init* is a set of $(Stat_L, Dyn_L, \Theta, V)$-updates (*initializations*) containing at most the variable π, $NF(lhs) = f(t_1, \ldots, t_n)$ for an $f \in \Delta$ and Ξ-terms $t_1, \ldots, t_n$, and $NF(rhs) \in T(\Xi)$ for each $lhs := rhs \in Init$, and
(O5) *Trans* is a set of closed $(Stat_L, Dyn_L, \Theta, V)$-rules (the *transitions*).

The *ASM for* π *defined by* $\mathbb{A}_L$ is the ASM $\mathcal{A}_\pi = (\Sigma_\pi, Q_\pi, S_\pi, \rightarrow_\pi, I_\pi)$ defined by the following properties (P1)–(P5):

(P1) $S_\pi = U$.
(P2) $\Sigma_\pi = \Delta_\pi \cup \Xi$.
(P3) Any $q \in Q_\pi$ is an algebra with $q|_\Xi = \mathcal{X}$ and $[\![t]\!]_q \in [\![B_\pi]\!]_\mathcal{X}$ for any $B \in S_\pi$.
(P4) $i \in I$ iff $[\![NF(lhs)]\!]_i = [\![NF(rhs)]\!]_i$ for all $lhs := rhs \in Init$.
(P5) $\rightarrow_\pi$ is defined by a set of transition rules obtained from *trans* in the following way: For any instruction $f(t_1, \ldots, t_n) \in INSTR_\pi$ and any rule

$$
\begin{array}{ll}
IP = f(x_1, \ldots, x_k) \rightsquigarrow & \mathbf{if}\ cond_1\ \mathbf{then}\ Updates_1 \\
& \mathbf{elsif}\ cond_2\ \mathbf{then}\ Updates_2 \\
& \vdots \\
& \mathbf{elsif}\ cond_m\ \mathbf{then}\ Updates_m \\
& \mathbf{else}\ Updates_0
\end{array}
$$

in *Trans*, the ASM $\mathcal{A}_\pi$ has the transition rule

$$
\begin{array}{l}
\mathbf{if}\ IP = f(t_1, \ldots, t_k)\ \mathbf{then} \\
\qquad \mathbf{if}\ NF(\sigma(cond_1))\ \mathbf{then}\ NF(\sigma(Updates_1)) \\
\qquad \mathbf{elsif}\ NF(\sigma(cond_2))\ \mathbf{then}\ NF(\sigma(Updates_2)) \\
\qquad \vdots \\
\qquad \mathbf{elsif}\ NF(\sigma(cond_m))\ \mathbf{then}\ NF(\sigma(Updates_m)) \\
\qquad \mathbf{else}\ NF(\sigma(Updates_0))
\end{array}
$$

where $\sigma = [x_1/t_1]\dots[x_m/t_m]$. Here, σ and NF are extended to sets of updates, i.e. $\sigma(\mathit{Updates}) = \{\sigma(\mathit{lhs}) := \sigma(\mathit{rhs}) : \mathit{lhs} := \mathit{rhs} \in \mathit{Updates}\}$ and $NF(\mathit{Updates}) = \{NF(\mathit{lhs}) := NF(\mathit{rhs}) : \mathit{lhs} := \mathit{rhs} \in \mathit{Updates}\}$.

Thus, the operational semantics for a language L defines a family $(\mathcal{A}_\pi)_{\pi\in L}$ of abstract state machines.
Notation: $\mathbb{A}_L$ also denotes this family of ASMs. ■

Example 3 (Basic Block Graphs, continued) Figure 3 shows the initial states and the transitions of the ASM of the program π defined by the operational semantics of BB (cf. Appendix A.1). The definition of *eval* is applied in constructing the transition rules.

Each initial state $i \in I$ satisfies:

$$\begin{aligned}
[\![\mathit{inp}]\!]_i &= [\![\mathit{standard_input}]\!]_{\mathcal{X}}\\
[\![\mathit{out}]\!]_i &= [\![\langle\rangle]\!]_{\mathcal{X}}\\
[\![BP]\!]_i &= 0\\
[\![IP]\!]_i &= \mathit{readint}(\mathit{local}(\mathit{intconst}_1))\\
[\![\mathit{loc}]\!]_i &= [\![\mathit{bot_of_stack}]\!]_{\mathcal{X}}\\
[\![\mathit{glob}]\!]_i &= [\![\mathit{bot_of_stack}]\!]_{\mathcal{X}}
\end{aligned}$$

The transition rules are

if $IP = \mathit{readint}(\mathit{local}(\mathit{intconst}_1))$
then $\overline{\mathit{content}_8}(\mathit{loc} \oplus_A 8) := \mathit{hd}(\mathit{inp})$
$\quad\quad \mathit{inp} := \mathit{tl}(\mathit{inp})$
$\quad\quad PC := \mathit{next}(PC)$

and

if $IP = \mathit{condjump}(\mathit{intequal}(\mathit{local}(\mathit{intconst}_1), \mathit{intconst}_0), 0, 1)$
then if $\mathit{content}_8(\mathit{loc} \oplus_A 8) =_I 0$
then $BP := 0$
$\quad\quad PC := 0$
else $BP := 1$
$\quad\quad PC := 0$

Figure 3: Initial States and Transition Rules of the ASM for the Program in Figure 2

Computation sequences denote sequences of state transitions of $\mathcal{A}_\pi$. Formally, a *computation sequence of program* $\pi \in L$ is a finite or infinite sequence qq over Q_π satisfying the following conditions:

(B1) $qq = \langle q_i : i \in \mathbb{N}\rangle$ iff $q_0 \in I_\pi$ and $q_i \to_\pi q_{i+1}$ for all $i \in \mathbb{N}$ and
(B2) $qq = \langle q_i : 0 \le i \le n\rangle$ $(n \in \mathbb{N})$ iff $q_0 \in I_\pi$, $q_i \to_\pi q_{i+1}$ for all $0 \le i < n$ and q_n is a final state.

A computation sequence is *terminating* iff it is finite. We denote computation sequences by

$$q_0 \rightarrow_\pi q_1 \rightarrow_\pi q_2 \cdots$$

The *behavior* B_π *of program* $\pi \in L$ is the set of computation sequences of $\pi \in L$. π is *terminating* iff every sequence in B_π is finite. $B_\pi(i) = \{qq \in B_\pi : hd(qq) = i\}$ is the *behavior of* π *on initial state* $i \in I_\pi$. π is *deterministic in the strong sense* iff $|B_\pi(i)| = 1$ for all $i \in I$. π is *terminating on* $i \in I_\pi$ iff any computation sequence in $B_\pi(i)$ is terminating. A language L is *deterministic in the strong sense* iff each $\pi \in L$ is deterministic in the strong sense.

Example 4 (Basic Block Graphs, continued) Consider the BB-program in Figure 2 and its ASM in Figure 3. For any state $i \in I_\pi$, where $[\![inputstream]\!]_i = [\![\langle 0, \ldots \rangle]\!]_\mathcal{X}$ is the infinite sequence of 0, it is

$$B_\pi(i) = \{i \rightarrow q_1 \rightarrow q_1' \rightarrow q_2 \rightarrow q_2' \cdots\}.$$

Table 2 shows the interpretation of the dynamic functions in the states of the computation sequence. If $[\![inpstream]\!]_\mathcal{X}$ is a infinite sequence containing a value

	$[\![PC]\!]$	$[\![BP]\!]$	$[\![inp]\!]$	$[\![out]\!]$	$[\![loc]\!]$	$[\![glob]\!]$
$q_j,\ j \geq 1$	0	0	$[\![tl^{j-1}(inp)]\!]_\mathcal{X}$	$[\![\langle\rangle]\!]_\mathcal{X}$	$[\![bot\_of\_stack]\!]_\mathcal{X}$	$[\![bot\_of\_stack]\!]_\mathcal{X}$
$q_j',\ j \geq 1$	1	0	$[\![tl^{j}(inp)]\!]_\mathcal{X}$	$[\![\langle\rangle]\!]_\mathcal{X}$	$[\![bot\_of\_stack]\!]_\mathcal{X}$	$[\![bot\_of\_stack]\!]_\mathcal{X}$

$$[\![content_8(a)]\!]_{q_j} = \begin{cases} [\![0_I]\!]_\mathcal{X} & \text{if } [\![a]\!]_\mathcal{X} = [\![bot\_of\_stack \oplus_A 8]\!]_\mathcal{X} \\ [\![content_8(a)]\!]_i & \text{otherwise} \end{cases}$$

$$[\![content_8(a)]\!]_{q_j'} = [\![content_8(a)]\!]_{q_j}$$

Table 2: Interpretation of the States in $B_\pi(i)$

different from 0, i.e.

$$[\![inpstream]\!]_\mathcal{X} = [\![\langle \underbrace{0, \ldots, 0}_{n}, x, \ldots \rangle]\!]_\mathcal{X} \text{ for a } n \geq 0,$$

then

$$B_\pi'(i) = \{i \rightarrow q_1 \rightarrow q_1' \rightarrow \cdots \rightarrow q_n \rightarrow q_n \rightarrow q_{n+1}\}.$$

Table 3 shows the interpretation of the dynamic functions in the states of the computation sequence of $B_\pi'(i)$.
The state q_{n+1} is final. π does not terminate, but it terminates on initial states where $[\![inp]\!]_i$ is different from the infinite sequence of zeros. π is not deterministic in the strong sense. ■

	$[\![PC]\!]$	$[\![BP]\!]$	$[\![inp]\!]$	$[\![out]\!]$	$[\![loc]\!]$	$[\![glob]\!]$
q_j	0	0	$[\![tl^{j-1}(inp)]\!]_{\mathcal{X}}$	$[\![\langle\rangle]\!]_{\mathcal{X}}$	$[\![bot\_of\_stack]\!]_{\mathcal{X}}$	$[\![bot\_of\_stack]\!]_{\mathcal{X}}$
q'_j	1	0	$[\![tl^{j}(inp)]\!]_{\mathcal{X}}$	$[\![\langle\rangle]\!]_{\mathcal{X}}$	$[\![bot\_of\_stack]\!]_{\mathcal{X}}$	$[\![bot\_of\_stack]\!]_{\mathcal{X}}$
q_{n+1}	0	1	$[\![tl^{n}(inp)]\!]_{\mathcal{X}}$	$[\![\langle\rangle]\!]_{\mathcal{X}}$	$[\![bot\_of\_stack]\!]_{\mathcal{X}}$	$[\![bot\_of\_stack]\!]_{\mathcal{X}}$

$$[\![content_8(a)]\!]_{q_j} = \begin{cases} [\![0_I]\!]_{\mathcal{X}} & \text{if } [\![a]\!]_{\mathcal{X}} = [\![bot\_of\_stack \oplus_A 1_I]\!]_{\mathcal{X}} \\ [\![content_8(a)]\!]_i & \text{otherwise} \end{cases}$$

$$[\![content_8(a)]\!]_{q'_j} = \begin{cases} [\![0_I]\!]_{\mathcal{X}} & \text{if } j \neq n \text{ and } [\![a]\!]_{\mathcal{X}} = [\![bot\_of\_stack \oplus_A 1_I]\!]_{\mathcal{X}} \\ [\![x_I]\!]_{\mathcal{X}} & \text{if } j = n \text{ and } [\![a]\!]_{\mathcal{X}} = [\![bot\_of\_stack \oplus_A 1_I]\!]_{\mathcal{X}} \\ [\![content_8(a)]\!]_i & \text{otherwise} \end{cases}$$

Table 3: Interpretation of the States in $B'_\pi(i)$

Two states $q, q' \in Q_\pi$ are *Ω-equivalent*, denoted by $q \sim_\Omega q'$ iff $[\![f]\!]_q = [\![f]\!]_{q'}$ for all $f \in \Omega$. It is not hard to see that $\sim_\Omega$ is an equivalence relation on Q_π. $[q]_\Omega$ denotes the *Ω-equivalence class of state q*. $[Q]_\Omega$ is the set of all Ω-equivalence classes. $q \overset{*}{q}{}' \; q \sim_\Omega q'$ means that no interaction with the environment took place, e.g. there is no input/output operation during the state transitions from q to q'. Let qq be a computation sequence for $\pi \in L$. The *observable part ob_{qq} of qq* is a finite or infinite sequence of Ω-equivalence classes satisfying the following three conditions:

(B3) If $qq = \langle q_i : i \in \mathbb{N}\rangle$ and there is an increasing infinite sequence $\langle r_j : j \in \mathbb{N}\rangle$ such that $r_0 = 0$ and $q_h \sim_\Omega q_k$ for all $j \in \mathbb{N}$, $r_j \leq h, k < r_{j+1}$ then $ob_{qq} = \langle [q_{r_j}] : j \in \mathbb{N}\rangle$.

(B4) If $qq = \langle q_i : i \in \mathbb{N}\rangle$ and there is an increasing finite sequence $\langle r_1, \ldots r_n\rangle$ such that $r_0 = 0$, $q_h \sim_\Omega q_k$ for all $0 \leq j < n$, $r_j \leq h, k < r_{j+1}$, and $q_h \sim_\Omega q_k$ for all $h, k \geq r_n$ then $ob_{qq} = \langle [q_{r_1}], \ldots, [q_{r_n}]\rangle$.

(B5) If $qq = \langle q_1, \ldots, q_m\rangle$ and there is an increasing finite sequence $\langle r_1, \ldots r_n\rangle$ such that $r_0 = 0$, $r_n \leq m$, $q_h \sim_\Omega q_k$ for all $0 \leq j < n$, $r_j \leq h, k < r_{j+1}$, and $q_h \sim_\Omega q_k$ for all $r_n \leq h, k \leq m$ then $ob_{qq} = \langle [q_{r_1}], \ldots, [q_{r_n}]\rangle$.

Figure 4 visualizes the ideas of observable parts of a computation sequence. The

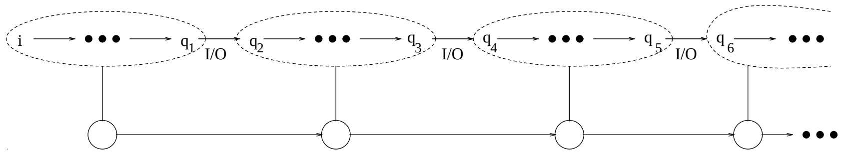

Figure 4: Observable Behavior

observable behavior of π is the set $OB_\pi = \{ob_{qq} : qq \in B_\pi\}$. The *observable behavior on an Ω-equivalence class $[i]_\Omega$ for an $i \in I$* is the set $OB_\pi([i]_\Omega) = \{qq \in OB_\pi : hd(qq) = [i]_\Omega\}$. A program π is *deterministic in the weak sense* if $|OB_\pi([i]_\Omega)| = 1$ for all $i \in I$. A language L is *deterministic in the weak sense* iff each $\pi \in L$ is deterministic in the weak sense. Two consequetive states in a computation sequence are either in the same Ω-equivalence class or in different Ω-equivalence classes. Hence, behaviours can be decomposed in maximal subsequences of Ω-equivalent states:

Lemma 3 Let $qq \in B_\pi$ be a computation sequence for a program $\pi \in L$. Then the following conditions hold:

(B6) If $qq = \langle q_i : i \in \mathbb{N}\rangle$ and $\{[q_i]_\Omega : i \in \mathbb{N}\}$ is infinite, then there exists an increasing infinite sequence $\langle j_i : i \in \mathbb{N}\rangle \in \mathbb{N}^*$ such that
  (B6-a) $q_{j_i} \not\sim_\Omega q_{j_{i+1}}$ for all $i \in \mathbb{N}$,
  (B6-b) $q_h \sim_\Omega q_{j_0}$ for all $0 \leq h \leq j_0$, and
  (B6-c) $q_h \sim_\Omega q_{j_i} \vee q_h \sim_\Omega q_{j_{i+1}}$ for all $i \in \mathbb{N}$, $j_i \leq h \leq j_{i+1}$.
(B7) If $qq = \langle q_i : i \in \mathbb{N}\rangle$ and $\{[q_i]_\Omega : i \in \mathbb{N}\}$ is finite, then there exists a finite increasing sequence $\langle j_0, \ldots, j_n\rangle \in \mathbb{N}^*$ such that (B6-b),
  (B7-a) $q_{j_i} \not\sim_\Omega q_{j_{i+1}}$ for all $0 \leq i < n$,
  (B7-b) $q_h \sim_\Omega q_{j_n}$ for all $h \geq j_n$, and
  (B7-c) $q_h \sim_\Omega q_{j_i} \vee q_h \sim_\Omega q_{j_{i+1}}$ for all $0 \leq i < n$, $j_i \leq h \leq j_{i+1}$.
(B8) If $qq = \langle q_0, \ldots, q_m\rangle$ then there exist a finite increasing sequence $\langle j_0, \ldots, j_n\rangle \in \mathbb{N}^*$ such that (B7-a), (B6-b),(B7-c) and
  (B8-a) $q_h \sim_\Omega q_{j_n}$ for all $j_n \leq h \leq m$.

Proof: The claim follows by induction from the fact that $q_i \sim_\Omega q_{i+1} \vee q_i \not\sim_\Omega q_{i+1}$ for all $q_i \to q_{i+1}$. ■

The sequences $\langle j_i : i \in \mathbb{N}\rangle$ and $\langle j_0, \ldots j_n\rangle$ in (B6)–(B8) are called *witnesses of the observable behavior of qq*. Lemma 3 immediately implies the

Corollary 4 Let $qq \in B_\pi$ be a computation sequence for $\pi \in L$ and jj be a witness of of the observable behavior of qq. Then: $jj = \langle j_i : i \in \mathbb{N}\rangle$ iff $ob_{qq} = \langle [q_{j_i}]_\Omega : i \in \mathbb{N}\rangle$ and $jj = \langle j_0, \ldots, j_m\rangle$ iff $ob_{qq} = \langle [q_{j_1}]_\Omega, \ldots, [q_{j_m}]_\Omega\rangle$. ■

Example 5 (Observable Behavior) Consider the operational semantics for BB in Example 3, the programs in Figure 5, and suppose $\Omega = \{inp, out\}$. The *writeint* instruction writes an integer to the output stream. The programs consist of a single basic block.

Let π and π' denote the left and right program, respectively. For any initial state $i \in I_\pi$, B_π contains the computation sequence

$$qq = i \to_\pi q_1 \to_\pi q_2 \to_\pi q_3 \to_\pi q_4$$

where $[\![inp]\!]_{q_j} = [\![inp]\!]_i$ for all $1 \leq j \leq 4$, $[\![IP]\!]_{q_4} = stop$,

$$[\![out]\!]_i = [\![out]\!]_{q_1} = [\![\langle\rangle]\!]_{\mathcal{X}}$$
$$[\![out]\!]_{q_2} = [\![out]\!]_{q_3} = [\![\langle 0_I\rangle]\!]_{\mathcal{X}}$$
$$[\![out]\!]_{q_4} = [\![\langle 0_I, 1_I\rangle]\!]_{\mathcal{X}}$$

```
0: intassign(local(intconst_0), intconst_0)
   intassign(local(intconst_1), intconst_1)
   writeint(cont(local(intconst_0)))
   writeint(cont(local(intconst_1)))
   stop
```

```
0: intassign(local(intconst_0), intconst_0)
   writeint(cont(local(intconst_0)))
   intassign(local(intconst_0), intconst_1)
   writeint(cont(local(intconst_0)))
   stop
```

Figure 5: Two Equivalent BB-programs

The definition of the remaining interpretations is left to the reader. It is easy to see that $OB_\pi = \{[i]_\Omega \to [q_3]_\Omega \to [q_4]_\Omega\}$ where

$$\llbracket out \rrbracket_q = \llbracket \langle\rangle \rrbracket_\mathcal{X} \text{ for all } q \in [i]_\Omega$$
$$\llbracket out \rrbracket_q = \llbracket \langle 0_I \rangle \rrbracket_\mathcal{X} \text{ for all } q \in [q_2]_\Omega$$
$$\llbracket out \rrbracket_q = \llbracket \langle 0_I, 1_I \rangle \rrbracket_\mathcal{X} \text{ for all } q \in [q_4]_\Omega$$

At $[i]_\Omega$ there is no output, at $[q_3]_\Omega$ 0 is written, and at $[q_4]_\Omega$ 1 is written. $\langle 0, 2, 4\rangle$ is a witness for the observable behavior of qq. This is not the only witness: $\langle 1, 3, 4\rangle$ is also a witness for the observable behavior of qq.
For any initial state $i' \in I_{\pi'}$, $B_{\pi'}$ contains the computation sequence

$$qq' = i \to_{\pi'} q'_1 \to_{\pi'} q'_2 \to_{\pi'} q'_3 \to_{\pi'} q'_4$$

where $\llbracket inp \rrbracket_{q'_j} = \llbracket inp \rrbracket_i$ for all $1 \leq j \leq 4$, $\llbracket IP \rrbracket_{q'_4} = stop$,

$$\llbracket out \rrbracket_i = \llbracket out \rrbracket_{q'_1} = \llbracket out \rrbracket_{q'_2} = \llbracket \langle\rangle \rrbracket_\mathcal{X}$$
$$\llbracket out \rrbracket_{q'_3} = \llbracket \langle 0_I \rangle \rrbracket_\mathcal{X}$$
$$\llbracket out \rrbracket_{q'_4} = \llbracket \langle 0_I, 1_I \rangle \rrbracket_\mathcal{X}$$

The definition of the remaining interpretations is left to the reader. It is easy to see that $OB_{\pi'} = \{[i']_\Omega \to [q'_3]_\Omega \to [q'_4]_\Omega\}$ where

$$\llbracket out \rrbracket_q = \llbracket \langle\rangle \rrbracket_\mathcal{X} \text{ for all } q \in [i']_\Omega$$
$$\llbracket out \rrbracket_q = \llbracket \langle 0_I \rangle \rrbracket_\mathcal{X} \text{ for all } q \in [q'_3]_\Omega$$
$$\llbracket out \rrbracket_q = \llbracket \langle 0_I, 1_I \rangle \rrbracket_\mathcal{X} \text{ for all } q \in [q'_4]_\Omega$$

At $[i']_\Omega$ there is no output, at $[q'_3]_\Omega$ 0 is written, and at $[q'_4]_\Omega$ 1 is written. $\langle 0, 3, 4\rangle$ is a witness of the observable behavior of qq' but $\langle 0, 2, 4\rangle$ is not a witness of the observable behavior of qq'. The two observable behaviors are equivalent. If the two programs are executed, then they produce the same output. In general, with the above choice of Ω, the observable behavior of BB-programs is their I/O-behavior. ■

We finish this subsection by defining semantics monomorphisms[2]. Let L be a language, and $\mathbb{A}_L$, $\mathbb{A}'_L$ be two operational semantics for L. A semantics monomorphism means that $\mathbb{A}_L$ is contained in $\mathbb{A}'_L$ in some sense. An *L-semantics monomorphism* $\zeta : \mathbb{A}_L \to \mathbb{A}'_L$ is a pair $(\bar{\phi}, \psi)$, where $\bar{\phi} : \Delta \cup \Xi \cup \Theta \to \Delta' \cup \Xi' \cup \Theta' \cup T(\Xi' \cup \Delta' \cup \Theta')$ is a injective mapping which can be canonically extended to an injective mapping ϕ on $T(\Xi \cup \Delta \cup \Theta)$, macros, updates, and transition rules, $\psi : \mathcal{X} \to \mathcal{X}'$ is a $\bar{\phi}$-monomorphism, and each of the following properties are satisfied:

- (SH1) $\bar{\phi}(f) = f$ for all $f \in \Sigma_L \cup \Gamma_L$, i.e., programs are preserved by ϕ.
- (SH2) If $f \in \Omega$ has positive arity, then $\bar{\phi}(f) \in \Omega'$. If $f \in \Omega$ is a constant, then there is an n-ary $g \in \Omega'$ and terms $t_1, \ldots, t_n \in T(\Xi' \cup \Delta' \cup \Theta')$ such that $\bar{\phi}(f) = g(t_1, \ldots, t_n)$.
- (SH3) If $f \in \Delta$ has positive arity, then $\bar{\phi}(f) \in \Delta'$. If $f \in \Delta$ is a constant, then there is an n-ary $g \in \Delta'$ and terms $t_1, \ldots, t_n \in T(\Xi' \cup \Delta' \cup \Theta')$ such that $\bar{\phi}(f) = g(t_1, \ldots, t_n)$.
- (SH4) If $f \in \Theta$ has positive arity, then $\bar{\phi}(f) \in \Theta'$. If $f \in \Theta$ is a constant, then there is an n-ary $g \in \Theta'$ and terms $t_1, \ldots, t_n \in T(\Xi' \cup \Delta' \cup \Theta')$ such that $\bar{\phi}(f) = g(t_1, \ldots, t_n)$.
- (SH5) $\phi(\mathit{Macros}_L) \subseteq \mathit{Macros}'_L$,$\phi(\mathit{Inits}_L) \subseteq \mathit{Inits}'_L$, and $\phi(\mathit{Trans}_L) \subseteq \mathit{Trans}'_L$.

The following theorem shows that semantics monomorphisms $\mathbb{A}_L \to \mathbb{A}'_L$ induce monomorphisms on the ASM for $\pi \in L$ defined by $\mathbb{A}_L$ into the ASM for π defined by $\mathbb{A}'_L$:

Theorem 2 Semantics Homomorphisms and ASM-Homomorphisms. Let L be language, $\mathbb{A}_L$, $\mathbb{A}'_L$ be two operational semantics for L, and $\zeta = (\bar{\phi}, \psi) : \mathbb{A}_L \to \mathbb{A}'_L$ an L-semantics monomorphism. Then for every $\pi \in L$ there exists a $\gamma : I \to I'$ such that $(\bar{\phi}, \psi, \gamma) : \mathcal{A}_\pi \to \mathcal{A}'_\pi$ is an ASM-monomorphism.

Proof: Let be $\pi \in L$. By Theorem 1, (SH5), (O3), (O5), and (P5), $\phi(\mathit{rule})$ is a transition rule of $\mathcal{A}'_\pi$ for every transition rule *rule* of $\mathcal{A}_\pi$. By Lemma 2, it is sufficient to prove that there is an injective mapping $\gamma : I \to I'$ satisfying (H1) for all $i \in I_1$. For each state $i \in I$ and term $t \in T(\Delta \cup \Xi)$, there is a term $t' \in T(\Xi)$ such that $[\![t]\!]_i = [\![t']\!]_{\mathcal{X}}$. Since ψ is a $\bar{\phi}$-monomorphism $\psi([\![t']\!]_{\mathcal{X}}) = [\![\phi(t')]\!]_{\mathcal{X}'}$. It remains to show that there is an initial state $i' \in I'$ such that $[\![\phi(t)]\!]_{i'} = [\![\phi(t')]\!]_{\mathcal{X}'}$. By (P4) i is initial iff $[\![NF(\mathit{lhs})]\!]_i = [\![NF(\mathit{rhs})]\!]_i$ for all $\mathit{lhs} := \mathit{rhs} \in \mathit{Inits}$. Since ζ is an L-semantics monomorphism, Init' and ϕ is injective, $\phi(NF(\mathit{lhs})) := \phi(NF(\mathit{rhs})) \in \mathit{Inits}'$ by Theorem 1. Thus any initial state $i' \in I'$ satisfies $[\![\phi(NF(\mathit{lhs}))]\!]_{i'} = [\![NF(\mathit{rhs})]\!]_{i'}$. Structural induction on the structure of terms t shows that there is an initial state $i' \in I'$ such that $[\![\phi(t)]\!]_{i'} = [\![\phi(t')]\!]_{\mathcal{X}'}$. ■

Semantics monomorphisms are used to embed a language L into a superlanguage $L' \supseteq L$ such that their behaviour is the same. The consequence is that the ASMs $\mathcal{A}_\pi \in \mathbb{A}_L$ and $\mathcal{A}'_\pi \in \mathbb{A}_{L'}$ are isomorphic. The motivation for requirement (SH3) is that some dynamic constants can be stored in registers. E.g. the dynamic constants *loc* and *glob* of the language *BB* defined in Appendix A.1 are stored in registers 1 and 2. Thus, $\bar{\phi}(\mathit{loc}) = \mathit{reg}(\mathit{1})$ and $\bar{\phi}(\mathit{glob}) = \mathit{reg}(\mathit{2})$. This is an arbitrary decision by the compiler writer.

[2] Homomorphisms are not needed and would complicate the definitions.

3.3 Correct Compilations

A *compiler* which compiles programs $\pi_1 \in L_1$ into a program $\pi_2 \in L_2$ implements a relation $\mathcal{C} : L_1 \to L_2$. Intuitively, $\mathcal{C}$ is correct if π_1 and π_2 have the same observable behavior if π_1 and π_2 are deterministic in the weak sense. For example the two programs in Figure 4 can be considered as a correct compilation of each other. This example shows that a correctness definition based on semantics monomorphisms would be too strong. Instead, we base the correctness definition on simulations, i.e. $\mathcal{A}_{\pi_2}$ simulates $\mathcal{A}_{\pi_1}$ in a sense similar to the notion of simulations used in complexity and computability theory. This subsection discusses an adequate formalization of these ideas (including the case of non-determinism in the weak sense), lifts the correctness definition from the observable behavior to the behavior, and discusses some general proof techniques.
In this subsection L_1 and L_2 are languages with operational semantics $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_2}$, respectively. To distinguish their components we index them with 1 and 2, respectively. $\mathcal{A}_{\pi,k}$ is the ASM for $\pi \in L_k$ defined by $\mathbb{A}_{L_k}$, $k = 1, 2$.

Definition 5 (ρ-Simulation of Computation Sequences) Let qq_1 and qq_2 be computation sequences of π_1 and π_2 respectively, and $\rho : [Q_{\pi,2}]_{\Omega_2} \to [Q_{\pi,1}]_{\Omega_1}$ an injective mapping. qq_2 *ρ-simulates* qq_1 iff either both computation sequences are terminating or both sequences are non-terminating, the observable parts of qq_1 and qq_2 have the same length, and the following conditions are satisfied:

(S1) If $ob_{qq_1} = \langle [q_i]_{\Omega_1} : i \in \mathbb{N}\rangle$ and $ob_{qq_2} = \langle [q'_i]_{\Omega_2} : i \in \mathbb{N}\rangle$ then $\rho([q'_i]_{\Omega_2}) = [q_i]_{\Omega_1}$ for all $i \in \mathbb{N}$.

(S2) If $ob_{qq_1} = \langle [q_1]_{\Omega_1}, \ldots, [q_n]_{\Omega_1}\rangle$ and $ob_{qq_2} = \langle [q'_1]_{\Omega_2}, \ldots, [q'_n]_{\Omega_2}\rangle$ then $\rho([q'_i]_{\Omega_2}) = [q_i]_{\Omega_1}$ for all $1 \le i \le n$. ■

If ρ is bijective, then qq_2 and qq_1 are ρ-observable equivalent.

Since ρ is not necessarily surjective, the fact that qq_2 ρ-simulates qq_1, does not imply that qq_1 ρ^{-1}-simulates qq_2. However, if ρ is bijective this implication holds. Figure 6 illustrates Definition 5. The notion of ρ-simulations is transitive.

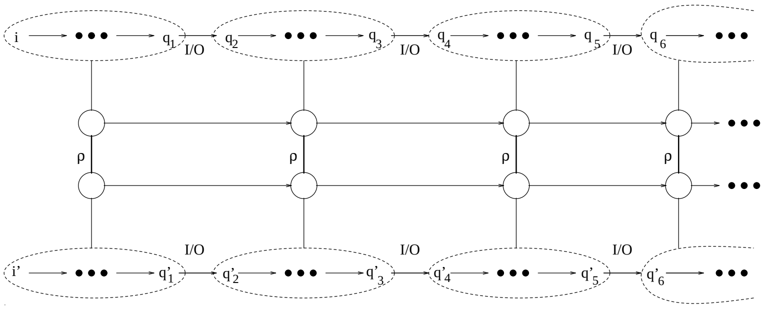

Figure 6: ρ-Simulation of Computation Sequences

Lemma 6 Let be qq_1, qq_2, and qq_3 be computation sequences of π_1, π_2, and π_3, respectively. Furthermore let $\rho' : [Q_{\pi_3}]_{\Omega_3} \to [Q_{\pi_2}]_{\Omega_2}$ and $\rho : [Q_{\pi_2}]_{\Omega_2} \to [Q_{\pi_1}]_{\Omega_1}$ be injective mappings. If qq_3 ρ'-simulates qq_2 and qq_2 ρ-simulates qq_1, then qq_3 $(\rho \circ \rho')$-simulates qq_1.

Proof: The proof is straightforward and left to the reader. ∎

Example 6 (Example 5, continued) The computation sequences of the programs π_1 and π_2 are observable equivalent. The relation ρ is the identity. ∎

Correct compilers preserve the termination properties of the program to be compiled. For simplicity we do not consider abnormal termination due to resource limitations of the target machine in this article. However, it is not difficult to extend our notion of correctness taking into account limited resources. The observable behavior of the compiled program can be mapped injectivly into the observable behavior of the corresponding uncompiled program.

Definition 7 (Compiler Correctness) *$\pi_2 \in L_2$ is a correct compilation of* $\pi_1 \in L_1$ iff there is an injective mapping $\rho : [Q_{\pi_2}]_{\Omega_2} \to [Q_{\pi_1}]_{\Omega_1}$ such that the following conditions are satisfied:

(CC1) For each $i_2 \in I_{\pi_2}$ and $qq_2 \in B_{\pi_2}(i_2))$, there is an $i_1 \in \rho([i_2]_{\Omega_2})$ and a $qq_1 \in B_{\pi_1}(i_1)$ such that qq_2 ρ-simulates qq_1.
(CC2) $\rho([I_{\pi_2}]_{\Omega_2}) = [I_{\pi_1}]_{\Omega_1}$.

A compiler $\mathcal{C} : L_1 \to L_2$ is *correct* w.r.t. $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_2}$, iff for all $\pi_1 \in L_1$ every $\pi_2 \in \mathcal{C}(\pi_1)$ is a correct compilation of π_1. ∎

Lemma 6 implies immediately the

Theorem 3 Vertical Decomposition. *Let L_1, L_2 and L_3 be languages and $\mathcal{C}_1 : L_1 \to L_2$ and $\mathcal{C}_2 : L_2 \to L_3$ be correct w.r.t. $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_2}$, and $\mathbb{A}_{L_2}$ and $\mathbb{A}_{L_3}$, respectively. Then $\mathcal{C}_2 \circ \mathcal{C}_1 : L_1 \to L_3$ is correct w.r.t. $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_3}$.*

Theorem 3 allows the use of traditional compiler architectures for the construction of correct compilers using ASMs. Any intermediate language used in a compiler can be introduced by proving just the correctness of the compilation of one intermediate language to another.
Definition 7 implies several properties.

Lemma 8 Let π_2 be a correct compilation of π_1. Then there is an injective mapping $\rho : [Q_{\pi_2}]_{\Omega_2} \to [Q_{\pi_1}]_{\Omega_1}$ such that (CC2) and the following properties are satisfied:

(CC3) For each $\langle [q'_i]_{\Omega_2} : i \in \mathbb{N}\rangle \in qq \in OB_{\pi_2}$: $\langle \rho([q'_i]_{\Omega_2}) : i \in \mathbb{N}\rangle \in OB_{\pi_2}$.
(CC4) For each $\langle [q'_1]_{\Omega_2}, \ldots, [q'_n]_{\Omega_2}\rangle \in qq \in OB_{\pi_2}$: $\langle \rho([q'_1]_{\Omega_2}), \ldots, \rho([q'_n]_{\Omega_2})\rangle \in OB_{\pi_2}$
(CC5) If π_1 terminates on every $i_1 \in I_{\pi_1}$, then π_2 terminates on every $i_2 \in I_{\pi_2}$.

Proof: Let π_2 be a correct compilation of π_1. By Definition 7, there is an injective mapping $\rho : [Q_{\pi_2}]_{\Omega_2} \to [Q_{\pi_1}]_{\Omega_1}$ satisfying (CC1) and (CC2). We show that ρ also satisfies (CC3)–(CC5). (CC3) follows directly from (CC1) and (S1). It also easy to see that (CC1) and (S2) imply (CC4). Suppose there is an infinite computation sequence $qq' \in B_{\pi_2}$ (i.e. π_2 does not terminate on every $i_2 \in I_{\pi_2}$). By (CC1), there is a computation sequence $qq \in B_{\pi_1}$ such that qq' ρ-simulates qq. Then, by Definition 5, qq is non-terminating, i.e. π_1 does not terminate on an $i_1 \in I_1$. ■

Remark: The converse of Lemma 8 is not true. E.g. suppose that π_2 does not terminate on a state $i_2 \in I_{\pi_2}$. Then there is an infinite computation sequence $qq' = \langle q'_i : i \in \mathbb{N}\rangle$. Suppose further, that $ob_{qq'}$ is terminating (cf. (B4)). Lemma 8 only ensures that there is a mapping ρ and a computation sequence $qq \in B_{\pi_1}$ such that (S2) is satisfied. Obviously, ob_{qq} is finite (cf. (CC4)). However, (CC2)–(CC5) do not exclude that qq is non-terminating. If the latter happens for all relations ρ satisfying (CC2)–(CC5), then π_2 cannot be a correct compilation of π_1. ■

A compiler defines a relation ρ on states not on its equivalence classes. This relation ρ must induce a function ρ on Ω-equivalence classes satisfying Definition 7.

Theorem 4 Necessary and Sufficient Condition for ρ-Simulation. π_2 is a correct compilation of π_1 iff there is a relation $\rho \subseteq Q_{\pi_1} \times Q_{\pi_2}$ satisfying

(CC6) $\forall (q_1, q'_1), (q_2, q'_2) \in \rho : q_1 \sim_{\Omega_1} q_2 \Leftrightarrow q'_1 \sim_{\Omega_2} q'_2$,
(CC7) $\forall [q_2] \in [Q_{\pi_2}]_{\Omega_2} \exists q' \in [q_2], q \in Q_{\pi_1} : (q, q') \in \rho$, and
(CC8) $\forall i \in I_{\pi_1} \exists q \in [i]_{\Omega_1}, q' \in Q_{\pi_2} : (q, q') \in \rho \wedge \exists i' \in I_{\pi_2} : q' \sim_{\Omega_2} i'$,

such that for any $qq' \in B_{\pi_2}$ there exist $qq \in B_{\pi_1}$ and witnesses jj and ll for the observable behavior of qq and qq', respectively, satisfying the following conditions:

(CC9) If $ll = \langle l_i : i \in \mathbb{N}\rangle$, then $jj = \langle j_i : i \in \mathbb{N}\rangle$ and $(q_{j_i}, q_{l_i}) \in \rho$ for all $i \in \mathbb{N}$.
(CC10) If $ll = \langle l_0, \dots, l_m\rangle$, then $jj = \langle j_0, \dots, j_m\rangle$ and $(q_{j_i}, q_{l_i}) \in \rho$ for all $i = 0, \dots, m$.
(CC11) qq' is terminating iff qq is terminating.

Proof: "⇐":Define $\bar\rho \subseteq [Q_{\pi_1}]_{\Omega_1} \times [Q_{\pi_2}]_{\Omega_2}$ by $([q]_{\Omega_1}, [q']_{\Omega_2}) \in \bar\rho$ iff $(q_1, q_2) \in \rho$. It is easy to see that (CC6) and (CC7) imply that $\bar\rho$ is an injective mapping $[Q_{\pi_2}]_{\Omega_2} \to [Q_{\pi_1}]_{\Omega_1}$. Furthermore, (CC8) implies that $\bar\rho$ satisfies (CC2). Let be $qq' \in B_{\pi_2}$. Then there exist $qq \in B_{\pi_1}$ and witnesses jj and ll for the observable behavior of qq and qq', respectively, such that (CC9)–(CC11) are satisfied. It remains to show that qq' $\bar\rho$-simulates qq. By (CC11) qq is terminating iff qq' is terminating. Suppose $qq' = \langle q'_i : i \in \mathbb{N}\rangle$. Then $qq = \langle q_i : i \in \mathbb{N}\rangle$ is non-terminating. Suppose further that ll is infinite. By the definition of witnesses $ob_{qq'} = \langle [q'_{l_i}]_{\Omega_2} : i \in \mathbb{N}\rangle$. By (CC9), jj is also infinite, implying $ob_{qq} = \langle [q_{j_i}]_{\Omega_2} : i \in \mathbb{N}\rangle$. (CC9) implies that $\bar\rho([q'_{l_i}]_{\Omega_2}) = [q_{j_i}]_{\Omega_2}$ for all $i \in \mathbb{N}$. The cases that qq is non-terminating and ll is finite, and the case that qq is finite imply (CC10) by a similar reasoning.

"⇒": Let be $qq' \in B_{\pi_2}$. Then there is a $qq \in B_{\pi_1}$ such that qq' $\bar\rho$-simulates qq (by (CC1)). Thus, (CC11) is satisfied. Consider the case $ob_{qq'} = \langle [\bar q'_i]_{\Omega_2} : i \in \mathbb{N}\rangle$.

Then (S1) implies that $ob_{qq} = \langle \bar{\rho}([\bar{q}'_i]_{\Omega_2}) : i \in \mathbb{N}\rangle$. Let ll and jj be witnesses of the observable behavior of qq' and qq, respectively. Then $q'_{l_i} \in [\bar{q}_i]_{\Omega_2}$ and $q_{j_i} \in \bar{\rho}([\bar{q}'_i]_{\Omega_2})$ for all $i \in \mathbb{N}$. ρ must be defined such that $(q_{j_i}, q_{l_i}) \in \rho$ for all $i \in \mathbb{N}$. Obviously, this definition satisfies (CC9). The case $ob_{qq'} = \langle[\bar{q}_0], \ldots [\bar{q}_m]\rangle$ proves analogously (CC10). It remains to show that ρ satisfies (CC6)–(CC8). Obviously $(q, q') \in \rho$ implies $([q]_{\Omega_1}, [q']_{\Omega_2}) \in \rho$. Thus, the fact that $\bar{\rho}$ is an injective mapping immediately implies (CC6) and (CC7). It is also easy to see that (CC1) implies (CC8). ■
Figure 7 illustrates Theorem 4.

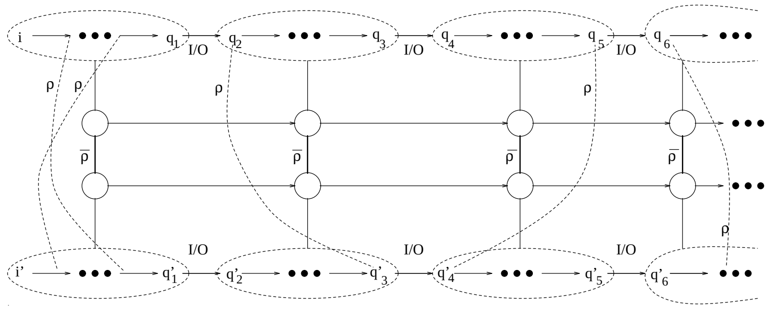

Figure 7: Theorem 4

Remark: ρ has to be implemented by a compiler. By properties (CC6)–(CC10), ρ ensures that at least the observable behavior of the ASM $\mathcal{A}_\pi$ is preserved by the ASM $\mathcal{A}_{\pi'}$. The relation ρ used in compilers is usually more specific: it contains memory mapping, the relation between instructions of π and π' etc. However it is hard to define ρ for real programming languages and machine languages explicitly. We define ρ as a composition of several explicitly defined relations (cf. Definition 21 in section 5). For the following discussions (until Definition 21), the precise definition of ρ is not important. ■
We say that qq' *ρ-simulates* qq if ρ is a relation satisfying (CC6)—(CC10).

Theorem 5 Horizontal Decomposition. Let $\pi_1 \in L_1$ and $\pi_2 \in L_2$, and $\rho \subseteq Q_{\pi_1} \times Q_{\pi_2}$ be a relation satisfying (CC6)–(CC8), $\rho^{-1}(I_{\pi_2}) \subseteq I_{\pi_1}$, and $\rho(F_{\pi_2})^3 \subseteq F_{\pi_1}$. Suppose that for all $(q_1, q'_1) \in \rho$ and states $q'_2 \in \rho(Q_{\pi_2})$ satisfying

(HD1) $q'_1 \overset{+}{\to}_{\pi_2} q'_2$ and $q' \sim_{\Omega_2} q'_1 \vee q' \sim_{\Omega_2} q'_2$ for all states q' such that $q'_1 \overset{*}{\to}_{\pi_1} q' \overset{*}{\to}_{\pi_1} q_2$,

there is a state $q_2 \in Q_{\pi_1}$ such that $(q_2, q'_2) \in \rho$, $q'_1 \sim_{\Omega_2} q'_2$ implies $q_1 \sim_{\Omega_1} q_2$,

(HD2) $q_1 \overset{+}{\to}_{\pi_1} q_2$, and $q \sim_{\Omega_1} q_1 \vee q \sim_{\Omega_1} q_2$ for all states q such that $q_1 \overset{*}{\to}_{\pi_1} q \overset{*}{\to}_{\pi_1} q_2$.

[3] Remind that $F\pi$ are the final states of π

Then π_2 is a correct compilation of π_1.

Proof: We first show that

(HD3) for any prefix $qq' = \langle q'_0, \ldots, q'_n \rangle$ of a computation sequence of $\mathcal{A}_{\pi_2}$ satisfying $q'_n \in \rho(Q_{\pi_1})$ there is a prefix $qq = \langle q_0, \ldots, q_m \rangle$ of a computation sequence of $\mathcal{A}_{\pi_1}$ satisfying $(q_m, q'_n) \in \rho$ and (CC10).

We prove (HD3) by induction on the length of qq'. If $qq' = \langle i' \rangle$ then there is an $i \in I_{\pi_1}$ such that $(i, i') \in \rho$, because $\rho^{-1}(I_{\pi_2}) \subseteq I_{\pi_1}$. Defining $qq = \langle i \rangle$ will do the job. Suppose now that $qq' = \langle q'_0, \ldots, q'_n \rangle$ for a $q'_n \in \rho(Q_{\pi_1})$. Let q'_m the last state before n with $q'_m \in \rho(Q_{\pi_1})$. By (HD1), $q'_i \sim_{\Omega_2} q'_m$ or $q'_i \sim_{\Omega_2} q'_n$ for all $i = m, \ldots, n$. By induction hypothesis there is a prefix $\widehat{qq} = \langle q_0, \ldots, q_l \rangle$ of a computation sequence of $\mathcal{A}_{\pi_1}$ satisfying (CC10) and $(q_l, q_m) \in \rho$. By (HD2), there are states $q_{l+1}, \ldots, q_r$ such that $q_l \to q_{l+1} \to \cdots \to q_r$, $(q_r, q'_n) \in \rho$, and $q_j \sim_{\Omega_1} q_l \vee q_j \sim_{\Omega_1} q_r$. q_r is chosen such that $q_r \sim_{\Omega_1} q_l$ if $q'_n \sim_{\Omega_2} q'_m$. The sequence $qq = \langle q_0, \ldots, q_l, q_{l+1}, \ldots, q_r \rangle$ is prefix of a computation sequence of $\mathcal{A}_{\pi_2}$. We finally have to show that (CC10) is satisfied for qq' and qq. Let $ll = \langle l_0, \ldots, l_s \rangle$ and $jj = \langle j_0, \ldots, j_s \rangle$ be witnesses for the sequences $\langle q'_0, \ldots, q_m \rangle$ and $\widehat{qq}$ satisfying (CC10). If $q'_n \sim_{\Omega_2} q'_m$, the same witnesses prove that (CC10) is satisfied. Otherwise, $ll = \langle l_0, \ldots, l_s, n \rangle$ and $jj = \langle j_0, \ldots, j_s, r \rangle$ prove that (CC10) is satisfied.
Suppose qq' is non-terminating and $ob_{qq'}$ is infinite. Then any prefix of qq' satisfies (HD3). The infinity of $ob_{qq'}$ implies (CC9). Suppose qq' is terminating. Then the last state of qq is final which implies that the last state of qq is also final, i.e. qq is terminating. Finally, assume that qq' is non-terminating. By taking a prefix qq'' of qq which is large enough, it is easy to prove that for every $n \in N$, there is a prefix $\widehat{qq}$ satisfying (HD3) of a computation sequence qq satisfying (CC9) and (CC10) such that the length of $\widehat{qq}$ is larger than n, i.e. q_1 is non-terminating. Thus (CC11) also holds. ■
Figure 8 illustrates the idea behind Theorem 5.

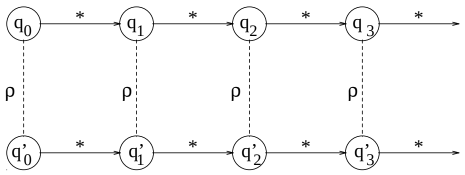

Figure 8: Horizontal Decomposition

Remark: In a compiler sequences s of L_1 instructions are transformed locally (using global information) into sequences s' of L_2-instructions. The basic idea to prove the correctness of compilation by proving (HD1)–(HD3) for the state transitions induced by s and s'. However, we will see that (HD1)–(HD3) cannot always be ensured without some additional assumptions. ■

3.4 Closely Related Languages

The basic idea in practical compilers is to introduce a sequence of intermediate languages $IL_0, \ldots, IL_n$ and to compile correctly IL_i-programs to IL_{i+1}-programs $i = 0, \ldots, n-1$. This approach leads to a correct compiler by Theorem 3. The reason for the decomposition into intermediate languages is that the correctness of $\mathcal{C}_\rangle : IL_i \to IL_{i+1}$ is easier to prove and to implement, if the languages IL_i and IL_{i+1} are closely related. This subsection formalizes the notion of closely related languages.
Informally, two languages L_1 and L_2 are closely related, if they either there is a one-to-one relation between their control structures (*control flow related*), or there is a one-to-one relation between their instruction set (*instruction set related*). A compiler $\mathcal{C} : L_1 \to L_2$ can then focus on either mapping the instructions set while preserving the control flow or mapping the control flow while preserving the instruction set. In this sense, the pair of languages BB and BB_α is an example of the former and the pair of languages BB_α and L_α of the latter (cf. appendix A). Since our goal is to prove correctness of compilers, we cannot ignore the operational semantics in this definition.
We first define control flow related languages. Informally, a language L_1 is control flow related to a language L_2 iff additionally to the above properties, it is possible to define an operational semantics of L_1 using the state space of L_2. This new operational semantics is defined such that it is an image of a L_1-operational semantics monomorphism. The consequence is that it is possible to run L_1-programs on the state space of L_2. This approach allows to extend L_1 by L_2-instruction and to describe the compilation $\mathcal{C} : L_1 \to L_2$ as source-to-source compilation.
The state space is usually divided into two parts: Dynamic functions occurring in the instruction pointer IP are related to the control flow (CR1). The other dynamic functions represent the state of the memory. Since the control flow of L_1 corresponds to a subset of the control flow of L_2, the dynamic functions occurring in IP_{L_1} have to correspond uniquely to the dynamic functions occurring in IP_{L_2} (CR2). The observable functions of L_1 must correspond to the observable functions of L_2 (CR3).

Definition 9 (Control Flow Related Languages) Let L_1 and L_2 be two languages with an operational semantics $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_2}$, respectively, $\Phi = (\Xi_2 \setminus \Sigma_2) \cup \Sigma_1$, and $U = (U_2 \setminus S_{L_2}) \cup S_{L_1}$. L_1 is *control flow related* to L_2 iff $\Delta_2 \cap \Phi = \emptyset$, $\Gamma_{L_1} = \Gamma_{L_2}$, there is an injective mapping $\bar{\phi} : \Delta_1 \cup \Xi_1 \to \Delta_2 \cup \Xi_2 \cup T(\Delta_2 \cup \Phi)$, a Φ-algebra $\mathcal{X}$, and a $\bar{\phi}$-monomorphism $\psi : \mathcal{X}_1 \to \mathcal{X}$ such that (SH2), (SH3), and the following conditions are satisfied:

(CR1) For every $f : T_1 \times \cdots \times T_k \to T_{k+1} \in \Sigma_{L_1} \cup \Gamma_{L_1}$, it is $\bar{\phi}(f) = f$ and $\bar{\phi}(T_i) = T_i$ for $i = 1, \ldots, k+1$.

(CR2) For every $f : T_1 \times \cdots \times T_k \to T \in \Delta_{L_1}$ occurring in IP_1, it is $\bar{\phi}(f) = f$ and $\bar{\phi}(T_i) = T_i$ for $i = 1, \ldots, k+1$. Furthermore $\phi(IP_1) = IP_1 = IP_2$.

(CR3) $\mathcal{X}|_{\Sigma_{L_1} \cup \Gamma_{L_1}} = \mathcal{I}_{L_1}$, and $[\![f]\!]_{\mathcal{X}} = [\![f]\!]_{\mathcal{X}_2}$ for every $f \in \Phi \setminus (\Sigma_1 \cup \Gamma_{L_1})$. $\blacksquare$.

The condition (CR1) states that the interpretation of all static functions in Ξ_1 except those for defining the structure and control-flow of L_1-programs can be mapped by a $\bar{\phi}$-monomorphism to the interpretation of the static functions of L_2 which are not used for defining the structure and control-flow of L_2-programs. The following lemma states that L_1-programs can be "executed on the state space of L_2".

Lemma 10 (Interpretation of L_1 by L_2) Let L_1 and L_2 be two languages with operational semantics $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_2}$, respectively, such that L_1 is control-flow related to L_2. Let Φ, U, $\bar{\phi}$, ψ, and $\mathcal{X}$ be defined as in Definition 9, and ϕ be the canonical extension of $\bar{\phi}$ (together with $\bar{\phi}(f) = f$ for all $f \in M$. Let $\mathbb{A}'_{L_1} = (Stat', Dyn', M_1, V_1, \phi(Macros_1), \phi(Inits_1), \phi(Trans_1))$ where $Stat' = (\Phi, U, \mathcal{X})$ and $Dyn' = (\Delta_{L_2}, IP_1, \bar{\phi}(\Omega_1))$. Then $\mathbb{A}_{L_1}$ is an operational semantics for L_1 and $\zeta = (\bar{\phi}, \psi) : \mathbb{A}_{L_1} \to \mathbb{A}_{L_2}$ is an L_1-semantics monomorphism.

Proof: We first show that $\mathbb{A}'_{L_1}$ is an operational semantics for L_1. Since $\Gamma_{L_1} \cup \Sigma_{L_1} \subseteq \Phi$, $S_{L_1} \subseteq U'$, and (CR3), $Stat'$ is a static part of $\mathbb{A}'_L$. (CR2), (SH2) and $\Delta_2 \cap \Phi = \emptyset$ show that Dyn' is a $Stat'$-signature of the dynamic part of $\mathbb{A}'_L$. Obviously, each $\phi(lhs) \hat{=} \phi(rhs)$ is a $(Stat', Dyn', M, V)$-macro. Theorem 1 together with the fact that $Macros_1$ satisfies (O3) implies that $\phi(Macros_1)$ also satisfies (O3). The fact that $\phi(Inits_1)$ satisfies (O4) is proven analogously. Let $f(x_1, \ldots, x_n) \rightsquigarrow rhs$ a $(Stat_1, Dyn_1, M_1, V_1)$-rule. Since $\phi(f(x_1, \ldots, x_n) \rightsquigarrow rhs) = f(x_1, \ldots, x_n) \rightsquigarrow \phi(rhs)$, this pair satisfies (O1) and (O2) using $Stat'$ and Dyn' instead of $Stat_1$ and Dyn_1. Thus, $f(x_1, \ldots, x_n) \rightsquigarrow \phi(rhs)$ is a $(Stat', Dyn', M, V)$-rule which is closed iff $f(x_1, \ldots, x_n) \rightsquigarrow rhs$ is closed. Therefore (O5) holds for $\phi(Trans_1)$.
It remains to show that ζ is an L_1-semantics monomorphism. $\bar{\phi}$ is injective and ψ is an algebra monomorphism. (SH2) and (SH3) are satisfied by Definition 9.(CR1) implies (SH1). (SH4) is satisfied since $\bar{\phi}(f) = f$ for all $f \in M_1$. ■
Informally, if L_1 is instruction-set related to L_2 iff additionally to the one-to-one correspondence of the instruction set (CR5), the state space except those dynamic functions used for the instruction pointer is the same (CR4), the interpretation of all static functions of L_2 except those used for building programs from instructions is the same (CR6), and for any program $\pi \in L_1$, there exists a program $\pi' \in L_2$ such that the program π' executes the same sequences of instructions as π performing the same updates except those affecting the instruction pointer (CR7).

Definition 11 (Instruction-Set Related Languages) Let L_1 and L_2 be two languages with an operational semantics $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_2}$, respectively, Φ_1 be the signature of L_1-instructions, and Φ_2 the signature of L_2-instructions, Ψ_1 be the static functions not used by L_1 and Ψ_2 be defined analogously. L_1 is *instruction set related to* L_2 iff

- (CR4) $f \in \Delta_1 \cap \Delta_2$ for all f not occurring in IP_1 and IP_2, $\Omega_2 = \Omega_1$,
- (CR5) There is a bijective mapping $\phi : T(\Upsilon_2) \to T(\Upsilon_1)$ and an injective signature morphism $\sigma : \Psi_2 \to \Psi_1$,
- (CR6) an algebra monomorphism $\psi : \mathcal{X}_2|_{\Psi_2 \cup \Upsilon_2} \to \mathcal{X}_1|_{\Psi_1 \cup \Upsilon_1}$ such that $\psi([\![t]\!]_{\mathcal{X}_2}) = [\![\sigma(\phi(t))]\!]_{\mathcal{X}_1}$ for every $t \in T(\Psi_2 \cup \Upsilon_2)$, and

(CR7) there is a mapping $\gamma : T(\Sigma_{L_2} \cup \Gamma_{L_2}) \rightarrow 2^{T(\Sigma_{L_1} \cup \Gamma_{L_1})}$ such that $\bigcup_{\pi \in L_2} \gamma(\pi) = \{\pi : \pi \in L_1\}$ and for all $\pi' \in L_2$, $\pi \in \gamma(\pi')$, there is an injective mapping $\rho : Q_{\pi'} \rightarrow Q_\pi$ satisfying the following conditions:
(a) For all $q \in Q_{\pi'}$ and terms $t \in T(\Psi_2 \cup \Upsilon_2 \cup \Delta_2)$, $\psi([\![t]\!]_q) = [\![\sigma(t)]\!]_{\rho(q)}$. Especially, $\psi([\![IP_2]\!]_q) = [\![\sigma(IP_1)]\!]_{\rho(q)}$.
(b) $\rho(I_{\pi'}) \subseteq I_\pi$.
(c) For all $q, q' \in Q_{\pi'}$, $q \rightarrow_2 q'$ implies $\rho(q) \rightarrow_1 \rho(q')$. ■

The next lemma shows that it is possible to compile correctly L_1-programs into L_2 programs using the mapping ρ defined by (CR7).

Lemma 12 Let L_1 and L_2 be two languages with operational semantics $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_2}$, respectively, such that L_1 is instruction-set related to L_2. Then, for any $\pi \in L_1$, there is a $\pi' \in L_2$ such that π' is a correct compilation of π. Moreover, for any computation sequence $qq = \langle q_i : i \in \mathbb{N}\rangle$ of π', $\rho(qq) = \langle \rho(q_i) : i \in \mathbb{N}\rangle$ is a computation sequence of π' and for any computation sequence $qq = \langle q_0, \ldots, q_n\rangle$ of π', $\rho(qq) = \langle \rho(q_0), \ldots, \rho(q_n)\rangle$ is a computation sequence of π.

Proof: The second claim on computation sequences implies that π' is a correct compilation of π using Theorem 4. To prove the second claim choose a $\pi' \in L_2$ such that $\pi \in \gamma(\pi')$. The claim follows easily by induction on the length of the computation using (CR7). ■
We finish this subsection with summarizing the two definitions:

Definition 13 (Closely Related Languages) Let L_1 and L_2 be two languages with an operational semantics $\mathbb{A}_{L_1}$ and $\mathbb{A}_{L_2}$, respectively. L_2 is *closely related* to L_1 iff L_2 is control-flow related to L_1 or L_2 is instruction-set related to L_1. ■

4 Constructing Correct Compiler Back-Ends

In this section we derive an architecture for correct compiler back-ends.
We assume that a back-end compiles basic block graphs into machine programs where the target machine is a register machine with a limited number of registers, eventually of different type (e.g. the DEC-Alpha processor, appendix A.2). However, the concrete instruction set is not important for our considerations. Subsection 4.1 defines the class of intermediate languages and machine languages for which our approach works. This classes contain commonly used intermediate languages and machine languages for most of the commercial processors.
It turns out that machine languages are usually not closely related to intermediate languages. In subsection 4.2 we show how to construct a language BML such that the target language is instruction-set related to BML, and BML is control-flow related to the intermediate language, provided that the target language and intermediate language belongs to the above classes. The basic idea to construct BML is that BML-programs are basic block graphs, but contain already target machine instructions instead of intermediate language instructions. Thus a back-end is divided into two components: code selection for compiling the intermediate language to BML and the code linearization for compiling BML into the machine language TL.

Subsection 4.3 introduces a specification technique for specifying the transformations of intermediate language instructions to target machine instructions. This specification technique is based on BURS. However, it assigns registers when applying a term-rewrite rule. We show that the correctness of applying term-rewrite rules may depend on the register assignment. Therefore, a planning and normalization is added which annotate the intermediate language program with additional information (e.g. register assignment) and normalize the intermediate language program by source-to-source transformation (cf. Subsection 4.4). The term-rewrite rules are then applied conditionally based on the annotations of the intermediate language program.

4.1 Basic Block Languages and Typical Machine Languages

The definition of a basic block language captures the basic block structure, classifies the instruction set into jump and non-jump instructions, but leaves open the concrete instruction set. Thus, it captures a wide range of intermediate languages used in traditional compilers. The languages BB and BB_α described in Appendices A.1 and A.3 are examples of basic block languages.

Definition 14 (Basic Block Language) A language IL with the operational semantics $\mathbb{A}IL$ is a *basic block language* iff the following conditions are satisfied:

(BB1) There are sorts $BLOCK, LABEL, JUMP, EXPR \in S_L$, representing *basic blocks*, *labels*, *jump instructions*, and *expressions*, respectively. It holds $JUMP \sqsubseteq INSTR$.

(BB2) There are functions $newblock : LABEL \times INSTR^* \to BLOCK$, $makeprog : LABEL \times BLOCK^* \to PROG \in \Sigma_L$. These are the only functions in Σ_L with result sort $BLOCK$ and $PROG$, respectively.

(BB3) For each $f : T_1 \times \cdots \times T_k \to INSTR \in \Sigma_L$, it is $T_i \neq INSTR$ for all $i = 1, \ldots, k$. If there are functions $g : S_1 \times \cdots \times S_l \to T_i$ such that there is a $1 \leq j \leq l$ with $S_j = T_i$, then $T_i \sqsubseteq EXPR$.

(BB4) If $jump : T_1 \times \cdots \times T_k \to JUMP \in \Sigma_L$, then for all $i = 1, \ldots, k$ either $T_i \sqsubseteq EXPR$ or $T_i = LABEL$.

(BB5) There are functions $start : PROG \to LABEL$, $next : \mathbb{N} \to \mathbb{N}$, $block\_label : BLOCK \to LABEL$, $get\_block : LABEL \times PROG \to BLOCK$, $get\_instr : \mathbb{N} \times BLOCK \to INSTR \in \Gamma_{IL}$. The interpretation $\mathcal{I}_{IL}$ satisfies the equalities shown in Figure 9.

(BB6) If $wd_{IL}(prog)$ is true, then each basic block ends with a sequence of jump instructions. Any other instruction of a basic block must not be a jump instruction. Furthermore, for any label used in an instruction, there is exactly one basic block in the program with this label.

(BB7) There are sorts $ADDR$, $VALUE \in U_{IL}$. Their elements are called *addresses* and *values*, respectively.

(BB8) There are constants $BP : LABEL$, $PC : \mathbb{N} \in \Delta_{IL}$ (the *block pointer* and *program counter*) and a function $content : ADDR \to VALUE \in \Delta$ (the *memory*). Furthermore $IP = get\_instr(PC, block(BP, prog))$. For any $f : T_1 \times \cdots \times T_n \to T \in \Delta_{IL} \setminus \{PC, BP\}$ it is $T_i \in U_{IL} \setminus S_{IL}$.

(BB9) There is a function $eval : EXPR \to VALUE \in \Theta_{IL}$. For any $f : T_1 \times \cdots \times T_k \to T \in \Sigma$ with $T \subseteq EXPR$, there is a macro
$eval(f(x_1, \ldots, x_k)) \hat{=} F(g_1(x_1), \ldots, g_k(x_k))$
where $F : U_1 \times \cdots \times U_k \to VALUE \in \Delta_{IL} \cup \Xi_{IL}$ such that for all $i = 1, \ldots, k$ $T_i \sqsubseteq EXPR$ implies $U_i \sqsubseteq VALUE$ and $g_i = eval$.

(BB10) $Init_{IL}$ contains at least the updates $BP := start(prog)$ and $PC := 0$.

(BB11) For any instruction $jump : T_1 \times \cdots \times T_k \to JUMP$, if there is a transition rule $jump(x_1, \ldots, x_k) \rightsquigarrow rhs \in Trans_{IL}$, then for every $i = 1, \ldots, k$ rhs contains the updates $BP := x_i$ and $PC := 0$ iff $T_i = LABEL$.

$block(BP, prog)$ is the *current block*. IP_{IL} is the *current instruction*. ■

For all $l, l' : LABEL$, $i \in \mathbb{N}$, $blcks : BLOCK^*$, $instrs : INSTRS^*$:

$$\begin{aligned}
start(makeprog(l, blcks)) &= l \\
next(i) &= i + 1 \\
block_label(newblock(l, instrs)) &= l \\
get_block(l, makeprog(l', \langle\rangle)) &= \bot \\
get_instr(i, newblock(l, instrs)) &= instrs_i
\end{aligned}$$

$$get\_block(l, makeprog(l', blcks)) =$$
$$\begin{cases} hd(blcks) & \text{if } [\![block\_label(hd(blcks))]\!]_{\mathcal{I}_{IL}} = [\![l]\!]_{\mathcal{I}_{IL}} \\ get\_block(l, makeprog(l', tl(blcks))) & \text{otherwise} \end{cases}$$

Figure 9: Interpretation of the Control Flow defined by (BB5)

Remark: For most intermediate language used in compilers, it is possible to define a basic block structure according to Definition 14. (BB1) introduces the notion of basic blocks, labels and jumps and defines that jumps are special instructions. The interpretation of labels is left open. The intermediate language designer is free to chose any interpretation. The function *newblock* states that each basic block has a label and consists of a finite sequence of instructions. A program consists of a label and a list of basic blocks, formalized by *makeprog*. Informally, the label defines the block where the execution of the program starts. (BB3) states that instructions are defined non-recursively. However, their components may defined recursively. Any such recursively defined component is an expression. E.g. this allows that the right hand side of integer assignments are integer expressions. (BB4) states that a jump instruction can have only expressions or labels as arguments. This allows for example to define conditional jumps and jumps corresponding to case instructions. The meaning of a jump instruction is that under certain conditions (depending on the expressions), the control jumps to the first instruction of the block with a label which is an argument of the jump instruction (cf. also (BB11)).

start computes the label of the block to start the execution. (cf. also (BB10)). $get\_block(l, prog)$ defines the block with label l. This function is partial. If it is defined, *get_block* is unique (cf. (BB6)). $get\_instr(i, block)$ selects the i-th instruction of *block*. These definitions are only used for defining initializations and transitions. Requirement (BB4) ensures that there is no jump instruction jumping to the interior of a basic block.
Addresses may also be values. It might be even reasonable that labels and natural numbers are values. If the instruction set contains procedure calls, it is a reasonable way to store the label and natural number in the memory (cf. (BB8)) to continue after the return with the command following the call. Often the memory is the same as on the target machine. In this case, we use the same macros as the target machine (cf. Fig. 11). If the intermediate language allows indirect addresses, then the memory must be able to store addresses. However, these are requirements due to particular assumptions on the instruction set. The block pointer points to the block where the current instruction is executed and PC gives the index in the basic block of this instruction. (BB8) states that there are no other dynamic functions referring to the program. (BB9) states the there is an expression evaluation macro which is inductively defined over the structure of expressions. ■
The requirements to machine languages are similarly general. We just assume that there are registers, the program is stored somewhere in the memory of the machine, and the values which can be stored in the memory are fixed-length Bit-sequences. However, we allow that the machine instructions are able to deal with values represented by Bit-sequences of different lengths. The registers may be able to store values represented by longer Bit-Sequences. We require that the length is a multiple of the values which can be stored at the memory. This is a typical situation in assembler languages. In the machine language of the DEC-Alpha Processor Family described in Appendix A.2 for example, the memory stores bytes which are 8-Bit sequences while registers may store quads which are 64-Bit sequences. However, there are machine instructions which operate on 32-Bit sequences and 64-Bit sequences as well.

Definition 15 (Typical Machine Language) A language TL is a *typical machine language* if TL and its operational semantics $\mathbb{A}_{TL}$ satisfy the following conditions:

(ML1) There are sorts $ADDR, CELL \in S_{TL}$. Their interpretation is isomorphic to BIT^l and BIT^s, respectively, for a $s \in \mathbb{N}$, $l \in \mathbb{N}$. l is the *address size*, s is the *cell size*. We assume that l is an integral multiple of s.

(ML2) $\Sigma_{TL} = \left\{ \begin{array}{c} makeinstr : BIT^k \to INSTR \\ makeprog : ADDR \times INSTR^* \to PROG \end{array} \right\}$ for a $k \in \mathbb{N}$ where k is an integral multiple of s. k is the *instruction size*.

(ML3) There are functions $start : PROG \to ADDR$, $next : ADDR \to ADDR$, $addr\_instr : \mathbb{N} \times PROG \to ADDR$, $get\_instr : ADDR \times PROG \to INSTR \in \Gamma_{TL}$. The interpretation $\mathcal{I}_{TL}$ satisfies the equalities shown in Figure 10.

(ML4) $well\_defined_{TL}(makeprog(a, instrs))$ iff a is aligned and each instruction of *instrs* is valid.

(ML5) There is a sort $VALUE \in U_{TL}$ where $[\![VALUE]\!]_{\mathcal{X}_{TL}} = \biguplus_{i=1}^{r} [\![BIT^{j_i s}]\!]$ for a $r > 0$, $1 \leq j_1 < j_2 < \cdots < j_r$.

(ML6) Δ_{TL} contains at least the functions $PC : ADDR$, $content : ADDR \to CELL$, and $reg : BIT^m \to BIT^h$, where $m > 0$ and h is an integral multiple of the cell size s. h is the *register size*. Furthermore $IP = get\_instr(PC, prog)$. IP is called the *current instruction*. Ξ_{TL} contains at least the functions $\oplus_A : ADDR \times BIT^k \to ADDR$ and $\oplus_R : BIT^m \times BIT^m \to BIT^m$.

(ML7) Θ_{TL} contains at least the functions $content_i : ADDR \to BIT^{i \cdot s}$, $\overline{content}_i : ADDR \to BIT^{i \cdot s}$, $first\_cell_i : BIT^{is} \to BIT^s$,$last\_cells_i : BITS^{is} \to BITS^{(i-1)s}$ $1 \leq i \leq j_r$ where j_r is defined by (ML5), $reg_i : BIT^m \to BIT^{i \cdot h}$, $\overline{reg}_i : BIT^m \to BIT^{i \cdot h}$, $first\_word_i : BIT^{ih} \to BIT^h$, and $last\_words_i : BITS^{ih} \to BITS^{(i-1)h}$ for $1 \leq i \leq \lceil j_r s$. Figure 11 shows the defining macros.

(ML8) $Init_{ML}$ contains at least the update $PC := start(prog)$

(ML9) For every $\pi \in TL$ the ASM $\mathcal{A}_\pi$ defined by $\mathbb{A}_{ML}$ is deterministic.

A TL-instruction *instr* is a *jump*-instruction iff its transition rule contains an update $PC := t$ different from $PC := next(PC)$. ■

For all $a, a' : ADDR$, $instrs : INSTR^*$:

$$start(makeprog(a, instrs)) = a$$
$$[\![next(a)]\!]_{\mathcal{I}_{TL}} = [\![a \oplus_A k/s]\!]_{\mathcal{I}_{TL}}$$
$$addr\_instr(i, makeprog(a, instrs)) = a'$$

$$get\_instr(a, makeprog(a', instrs)) = \begin{cases} instrs_i & \text{if } addr\_instr(i, makeprog(a, instrs)) \\ \bot & \text{if for all } j \in \mathbb{N} : addr\_instr(i, makeprog(a, instrs)) \neq a \end{cases},$$

where $[\![a']\!]_{\mathcal{I}_{TL}} = [\![a \oplus_A (i-1) \cdot s]\!]_{\mathcal{I}_{TL}}$ and $(i-1) \cdot s$ is identified with the bit sequence of length k representing $(i-1)s$.

Figure 10: Interpretation of the Functions required by (ML3)

Remark: The sorts $ADDR$, $CELL$ are required, since the program is stored in the memory. $CELL$ represents the Bit sequences which can be stored in one memory cell. Usually, $CELL$ is isomorphic to bytes. However for being flexible, we cannot require that the size of a memory cell is one byte. The only requirement is that processors operate on sequences of bits. Often, the address size is an integral multiple of the cell size. *makeinstr* ensures that instructions are k-Bit sequences. Since these sequences must be stored in the memory, it is a reasonable

$$
\begin{aligned}
content_i(a) &\mathrel{\hat{=}} content(a) \circ content_{i-1}(a \oplus_A 1),\ \ i = 2, \ldots, j_r \\
content_1(a) &\mathrel{\hat{=}} (a) \\
reg_i(a) &\mathrel{\hat{=}} reg(a) \circ reg_{i-1}(a \oplus_R 1),\ \ i = 2, \ldots, \lceil j_r s/h \rceil \\
reg_1(a) &\mathrel{\hat{=}} reg(a) \\
\overline{content}_i(a) := x &\mathrel{\hat{=}} content(a) := first\_cell_i(x) \\
& \quad\ \overline{content}_{i-1}(a \oplus 1) := last\_cells_i(x),\ \ i = 2, \ldots, j_r \\
\overline{content}_1(a) := x &\mathrel{\hat{=}} content(a) := x \\
\overline{reg}_i(a) := x &\mathrel{\hat{=}} reg(a) := first\_word_i(x) \\
& \quad\ \overline{reg}_{i-1}(a \oplus 1) := last\_words_i(x),\ \ i = 2, \ldots, \lceil j_r s/h \rceil \\
\overline{reg}_1(a) := x &\mathrel{\hat{=}} reg(a) := x
\end{aligned}
$$

Figure 11: Macros required by (ML7)

assumption that k is a multiple of s. The first argument in *makeprog* is the lowest address where the program is stored in the memory. W.l.o.g. we assume that the instruction at this address is also the address where the execution starts (cf. (ML3) and (ML8)). The formalization of (ML4) is left to the reader. Alignment means that in order to store k Bits in memory cells of size s-Bits, the last r-Bits must be 0 for a $r > 0$. In some processors, not every Bit-sequence represents an instruction. Since we do not want to exclude this possibility, the second requirement is needed.
The function *start* computes the address where the execution starts. *addr_instr* is being defined such that a program $makeprog(a, instrs)$ is stored consecutively in the memory, starting with the address a. $get\_instr(a, prog)$ computes the instruction stored at address a. Many processors operate on values represented by bit sequences of different length. E.g. the DEC-Alpha processor family can operate on bit sequences of length 32, 64, and 128. However, these bit sequences are stored consecutively in the memory. Therefore it is reasonable to require that their length is a multiple of the cell size s (cf. (ML5)).
PC is the program counter. It contains the address of the instruction to be executed. *content* models the memory of the processor. *reg* models the registers of a processor. There are 2^m such registers. This requirement does not exclude that some of these registers are special (e.g. address registers, status registers). We assume that *reg* contains all the registers which can be addressed directly by the programmer. In general, it is not necessary to require that registers are addressed with m Bits. The instruction may determine which kind of register is chosen (e.g. accumulators or address registers). In this case, less than m Bits are sufficient to address the registers. E.g. the language L_α described in Appendix A.2 contains registers for storing $QUAD$s and floating point registers. The instructions determine which registers are used. For example, the instruction for adding floating point numbers use always the floating point registers. $\oplus_A$ is used to add relative addresses onto base addresses. $\oplus_R$ has the same meaning for register addresses. These functions are used to specify the macros required by (ML7).

The macro $content_i(a)$ is used to read i consecutive cells of the memory starting from a. The result is the concatenation of the bit sequences of these cells, i.e. a bit sequence of length is. Similarly, reg_i reads i consecutive registers. In both cases there may requirements that addresses and register addresses are aligned. However, these restrictions may exclude some instructions (cf.(ML4)). If a program is well-defined, no unaligned addresses are used. $\overline{content_i}(a)$ is used to store sequences of length is at i consecutive memory cells starting from a. The functions $\mathit{first\_cell}_i$ and $\mathit{last\_cells}_i$ select and delete the first s Bits from bit sequences of length is, respectively. These functions are auxiliary functions useful to define the above memory accesses. Similarly, $\mathit{first\_word}_i$ and $\mathit{last\_word}_i$ are auxiliary functions used to define the access to the registers. Their precise definition is straightforward and left to the reader. It is useful to have some other macros defining functions to shorten and extend Bit-sequences (cf. Appendix A.2). The latter may be signed and unsigned extensions. ■

Typical machine languages are usually not closely related to basic block languages. They cannot be control-flow related since (CR1) contradicts (BB5) and (ML5), and (CR2) contradicts (BB8) and (ML6). If the instruction set of a basic block language contains recursively defined expressions, then a typical machine language cannot be instruction-set related to the basic block language, since (ML2) implies that (CR5) is violated.

4.2 The Architecture

Since typical machine languages are usually not closely related to basic block languages, we introduce a further language BML (called *basic block machine language*) such that the machine language is closely related to BML and BML is closely related to the basic block language. Typical machine languages differ in their instruction set as well as in their control structures. Hence, BML must either keep the control structure of the intermediate using the instruction set of the machine language or keep the instruction set of the intermediate language using the control structures of the machine language. We decide to choose the former approach because this is commonly chosen in compilers.

Assumption: Let IL be a basic block language and TL be a typical machine language. For simplicity, we assume the following properties:

- (A1) $\Psi_{IL} = \Xi_{IL} \setminus (\Gamma_{IL} \cup \Sigma_{IL}) = (\Xi_{TL} \setminus (\Gamma_{TL} \cup \Sigma_{TL})) \cup \{ADDR, CELL)$, i.e. the signature is equal except the signature of programs.
- (A2) $[\![T]\!]_{\mathcal{X}_{IL}} = [\![T]\!]_{\mathcal{X}_{TL}}$ for every universe $T \in U_{IL} \cap \Psi_{IL}$ and $[\![t]\!]_{\mathcal{X}_{IL}} = [\![t]\!]_{\mathcal{X}_{TL}}$ for every $t \in T(\Psi_{IL})$, i.e. the interpretation of the sorts and Ψ_{IL}-terms is equal in both static algebras.
- (A3) $[\![LABEL]\!]_{\mathcal{X}_{IL}} = [\![ADDR]\!]_{\mathcal{X}_{TL}}$ and $ADDR \sqsubseteq VALUE$, i.e. labels are addresses and addresses are values.
- (A4) $\Omega_{IL} = \Omega_{TL}$, $content : ADDR \to CELL \in \Delta_{IL}$, any $f \in \Delta_{IL} \setminus (\Delta_{TL} \cup \{PC, BP\}$ is a constant $f : VALUE$. $\Theta_{TL} \setminus \{reg_i, \overline{reg}_i : BIT^m \to BIT^{ih}\} \subseteq \Theta_{IL}$, and $Macros_{IL}$ contains the corresponding definitions shown in Figure 11.
- (A5) Any target machine jump instruction contains exactly one jump target. ■

These assumptions are satisfied by the languages BB and L_α (cf. Appendices A.1 and (A.2). We now define .0the definition of basic block machine language BML

obtained from a basic block language IL and a typical machine language TL satisfying the above assumptions. Informally, BML builds basic blocks using TL-instructions instead of IL-instructions. Every static function $f \in \Xi_{IL}$ except those in the signature Υ_{IL} of instructions are interpreted equally by $\mathcal{X}_{IL}$ and $\mathcal{X}_{BML}$ if BML has the same instruction set as IL. If we would "replace" the sub-algebra $\mathcal{X}|_{\Upsilon_{IL}}$ by the trivial algebra and "replace" the sub-algebra $\mathcal{X}|_{\Upsilon_{BML}}$, then two resulting algebras would be the same. Before we define basic block machine languages, we formalize this "replacement": Let L be a basic block language with the signature Υ_L of instructions, the static signature Ξ_L and the static algebra $\mathcal{X}_L$. The *instruction set ignoring algebra of* $\mathcal{X}_L$ is the algebra $\mathcal{X}_{L,\bullet}$ with the properties shown in Figure 12. The replacement can then be formalized by requiring that $\mathcal{X}_{IL,\bullet} = \mathcal{X}_{BML,\bullet}$.

$$[\![T]\!]_{\mathcal{X}_{L,\bullet}} = \begin{cases} \{\bullet_T\} & \text{if } T \in \Upsilon_L \\ [\![T]\!]_{\mathcal{X}_L} & \text{if } T \in \Psi_L \\ \mathit{blocks} & \text{if } T = \mathit{BLOCK} \\ \mathit{progs} & \text{if } T = \mathit{PROG}_L \end{cases}$$

where $\mathit{blocks} = \{\mathit{newblock}(l, \langle\underbrace{\bullet, \ldots, \bullet}_{n}\rangle) : l \in [\![\mathit{LABELS}]\!]_{\mathcal{X}_{L,\bullet}}, n \in \mathbb{N}\}$ and $\mathit{progs} = \{\mathit{makeprog}(l, bb) : l \in [\![\mathit{LABELS}]\!]_{\mathcal{X}_{L,\bullet}} \wedge \exists n \in \mathbb{N}, b_1, \ldots, b_n \in \mathit{blocks} : bb = \langle b_1, \ldots, b_n\rangle\}$. For every $f : T_1 \times \cdots \times T_n \to T \in \Xi_L$:

$$[\![f]\!]_{\mathcal{X}_{L,\bullet}}(a_1, \ldots, a_n) =$$

$$\begin{cases} \bullet_T & \text{if } T \in \Upsilon_L \text{ and } a_i = \bullet_{T_i},\ i = 1, \ldots, n \\ [\![f]\!]_{\mathcal{X}_L}(a_1, \ldots, a_n) & \text{if } f \in \Psi_L \\ \mathit{newblock}(l, x) & \text{if } f = \mathit{newblock},\ l \in [\![\mathit{LABELS}]\!]_{\mathcal{X}_{L,\bullet}}, \\ & \text{and } x \text{ is a list of } \bullet_{\mathit{INSTR}_L} \\ \mathit{makeprog}(l, x) & \text{if } f = \mathit{makeprog},\ l \in [\![\mathit{LABELS}]\!]_{\mathcal{X}_{L,\bullet}}, \\ & \text{and } x \text{ is a list of elements of } \mathit{blocks} \\ \perp & \text{otherwise} \end{cases}$$

List operations are interpreted as usual

Figure 12: Instruction Set Ignoring Algebras of $\mathcal{X}_L$

Definition 16 (Basic Block Machine Language) Let IL be a basic block language with operational semantics $\mathbb{A}_{IL}$ and TL be a typical machine language with operational semantics $\mathbb{A}_{TL}$. A basic block language BML with operational semantics $\mathbb{A}_{BML}$ is the *basic block machine language obtained from IL and TL* iff the following conditions are satisfied:

(BM1) $\Upsilon_{BML} = \{\mathit{makeinstr} : \mathit{BIT}^k \to \mathit{INSTR}\} \cup \{\mathit{jump} : \mathit{BIT}^k \times \mathit{LABEL} \to \mathit{JUMP}\}$. *well_defined* is being defined such that $\mathit{makeinstr}(b)$ is an instruction iff b is a non-jump instruction and $\mathit{makeinstr}(b, L)$ is an instruction iff b is a jump instruction.

(BM2) $\Sigma_{BML} = \Sigma_{IL} \backslash \Upsilon_{IL} \cup \Upsilon_{BML}$, $\Gamma_{BML} = \Gamma_{IL}$, $S_{BML} = (S_{IL} \cup S_{TL}) \cap \Sigma_{BML}$, and $\mathcal{I}_{BML,\bullet} = \mathcal{I}_{\mathcal{IL},\bullet}$.

(BM3) $\Xi_{BML} = \Psi_{IL} \cup \Sigma_{BML} \cup \Gamma_{BML}$, $U_{BML} = \Xi_{BML} \cap (U_{IL} \cup U_{BML})$, and $\mathcal{X}_{BML,\bullet} = \mathcal{X}_{\mathcal{IL},\bullet}$.

(BM4) $\Delta_{BML} = \{PC : \mathbb{N}, BP : LABEL\} \cup (\Delta_{TL} \setminus \{PC\})$ and for any constant $f : VALUE \in \Delta_{IL} \setminus \Delta_{BML}$, there is a macro $f \hat{=} t$ and $\bar{f} := x \hat{=} t' := x$ for terms $t, t' \in T(\Psi_{BML} \cup \Delta_{BML})$ and variables $x \in V_{BML}$

(BM5) The set $Macros_{BML}$ contains all macros of $Macros_{TL}$, the macros of IL used for the initializations of the functions corresponding to the macros defined by (BM4), and the macros defined by (BM4).

(BM6) The set $Init_{BML}$ contains all initialization of $Init_{TL}$ except those with left hand side PC, it contains the initializations defined in (BB10), and it contains the initialization $\bar{f} := m$ iff $f := m \in Init_{IL}$ for all constants f defined as in (BM4).

(BM7) The set $Trans_{BML}$ consists of all transition rules of $Trans_{TL}$ for non-jump instructions, and a transition rule $jump(b, l) \rightsquigarrow rhs'$ for each jump instruction of TL where $b \rightsquigarrow rhs \in Trans_{TL}$ and rhs' is obtained from rhs by replacing the updates $PC := t$ where $t \neq next(PC)$ with $BP := l;\ PC := 0$. ■

(BM1) makes jump instructions explicit such that there is a one-to-one correspondence between *TL*-instructions and *BML*-instructions. (BM2) states that the instruction set of *IL* is replaced by the instruction set of *TL*, and that the interpretation of the control flow is the same as in *IL* except for instructions. (BM3) states the same for the rest of the static functions. (BM4) states that except for the dynamic functions used to refer to the program, the state space is the same as the state space of *TL*. (BM6) states that the same initializations as by *IL* are executed. (BM7) states that the transition rules are the same as in *TL* except for jump instructions where the jumps are based on changing the block pointer.
Such a basic block machine language will do the job:

Theorem 6. Let *IL* be a basic block language with operational semantics $\mathbb{A}_{IL}$, *TL* be a typical machine language with operational semantics $\mathbb{A}_{TL}$ and *BML* be the basic block machine language obtained from *IL* and *TL* with operational semantics $\mathbb{A}_{BML}$. Then *BML* is control-flow related to *IL* and *TL* is instruction-set related to *BML*.

Proof: We prove the first claim and leave the second to the reader. The key to the proof is the algebra $\mathcal{X}$ required by Definition 9. For our purpose the choice $\mathcal{X} = \mathcal{X}_{IL}$ will do the job. This is possible due to assumption (A1)–(A4). ψ is simply the identity. Furthermore, we define $\bar{\phi}(f) = f$ for all $f \in \Delta_{BML}$ and $\bar{\phi}(f) = t$ for all macros $f \hat{=} t$ introduced by (BM4). It is not hard to see that the properties (SH2), (SH3), (CR1), (CR2) and (CR3) are satisfied with these definitions. ■
The compilation of *IL*-programs to *BML*-programs is called *code selection* and the compilation of *BML*-programs to *TL*-programs is called the *code linearization*. We focus here on the former. The language BB_α is the merge of the languages *BB* and L_α. In particular, it uses the macros $loc \hat{=} reg_{quad}(1)$ and $glob \hat{=} reg_{quad}(2)$. The transition rules mentioned explicitly in Appendix A.3 are the jump instructions obtained by (BM7) from the instructions B, BR, and JMP of the DEC-Alpha assembly language.

4.3 Generation of Code-selection by Term-Rewriting

Term rewriting is commonly used in compiler back-end generators for the specification of the transformation to be performed by the code selection. Throughout the whole subsection we assume that *IL* is a basic block language with operational semantics $\mathbb{A}_{IL}$, *TL* is a target language with operational semantics $\mathbb{A}_{TL}$, and *BML* is the basic block machine language obtained from *IL* and *TL*. Additionally to assumptions (A1)–(A5), we assume that

(A6) the target language contains macros $f(x_1,\ldots,x_n)\hat{=}makeinstr(f\circ x_1\circ\cdots\circ x_n)$ for each operation code f which is not a jump instruction and macros $f(x_1,\ldots,x_n,l)\hat{=}jump(f\circ x_1\circ\cdots\circ x_n,l)$, where $x_i\in V\cup BIT^*$ and $l\in V\cup LABEL$. $IM\subseteq M_{TL}$ denotes the signature of these *instruction macros.*

This assumption allows to describe patterns of the machine instructions. In Appendix A.2 we used this technique to describe the machine instructions of the DEC-Alpha Assembly Language.
This subsection introduces a specification method based on term-rewriting for the compiling relation $\mathcal{CS}: IL\to BML$ for the code selection. Since transformation rules are applied locally, a *BML*-program is obtained by successively applying these rules until the program contains only *BML*-instructions. Hence, during this transformation process, programs may contain target machine as well as intermediate language instructions. Therefore, we extend *BML* with intermediate language instructions. Just adding the transition rules from *IL* will do the job. We extend expressions by registers such that intermediate language instructions can read and write registers. During this transformations, registers must be assigned to store values resulting from expression evaluations. Figure 13 shows the whole transformational process with the participating languages.

$$IL\ni\pi_0\in ML\rhd\underbrace{\pi_1\rhd\cdots\rhd\pi_{n-1}}_{\in ML}\rhd ML\ni\pi_n\in BML\rhd^*\pi'\in TL$$

$\pi\rhd^*\pi'$ is the linearization. $\rhd$ is the application of one transformation rule.

Figure 13: Transformations and Languages Used in Compiler Back-Ends

Definition 17 (Merged Language) A language $ML = merge(IL, BML)$ with operational semantics $\mathbb{A}_{ML}$ is a *merge* of *BML* and *IL* iff

(M1) $\Sigma_{ML}=\Sigma_{BML}\cup\Sigma_{IL}\cup\{Reg: BIT^m\to EXPR$, $\Gamma_{ML}=\Gamma_{IL}$, $\mathcal{I}_{ML}|_{\Sigma_{IL}\cup\Gamma_{IL}}=\mathcal{I}_{IL}$, $\mathcal{I}_{ML}|_{\Sigma_{BML}\cup\Gamma_{BML}}=\mathcal{I}_{BML}$, and wd_{BML} is extended such that also intermediate language instructions and expressions $Reg(n)$ are allowed.

(M2) $\Xi_{ML}=\Xi_{BML}\cup\Sigma_{IL}, \mathcal{X}_{ML}|_{\Sigma_{IL}\cup\Gamma_{IL}}=\mathcal{X}_{IL},\ \mathcal{X}_{ML}|_{\Sigma_{BML}\cup\Gamma_{BML}}=\mathcal{X}_{BML}$, and $U_{ML}=U_{IL}\cup U_{BML}$.

(M3) $\Delta_{ML}=\Delta_{BML}$, $\Omega_{ML}=\Omega_{BML}$, and $IP_{ML}=IP_{BML}$.

(M4) $\Theta_{ML} = \Theta_{IL} \cup \Theta_{BML}$, $V_{ML} = V_{IL} \cup V_{BML}$, $Macros_{ML} = Macros_{IL} \cup Macros_{BML} \cup \{eval(Reg(x)) \hat{=} reg(x)\}$, $Inits_{ML} = Inits_{IL} \cup Inits_{BML}$, and $Trans_{ML} = Trans_{IL} \cup Trans_{BML}$. ■

(M1) states that the instruction set contains instructions from IL as well as from BML, that the control flow is the same as in IL, and that the access to a register is an expression. (M2) states that the interpretation of static functions and universes extends the interpretations of the universes of IL and BML. We can embed IL and BML into the merged language ML:

Theorem 7. Let $ML = merge(IL, BML)$ be the merged language with operational semantics $\mathbb{A}_{ML}$. Then $IL \subseteq ML$, $BML \subseteq ML$, there is an IL-semantics monomorphism $\zeta_1 : \mathbb{A}_{IL} \rightarrow \mathbb{A}_{ML}$, and a BML-language monomorphism $\zeta_2 : \mathbb{A}_{ML} \rightarrow \mathbb{A}_{ML}$.

Proof: The property of sublanguages follows directly from (M1). We define the monomorphism ζ_1 by defining $\bar{\phi}(f) = f$ for all $f \in \Delta_{ML} \cup \Xi_{ML} \cup \Theta_{ML}$ and $\bar{\phi}(f) = NF_{ML}(f)$ for $f \in \Delta_{IL} \setminus \Delta_{ML}$. By (M3), $\bar{\phi}(f) = NF_{ML}(f)$ is the only possibility of $f \notin \Delta_{ML} \cup \Xi_{ML} \cup \Theta_{ML}$, by (BM4) this f must be a constant and a macro $f \hat{=} t$ exists. Using (M4), it is easy to prove that this definition satisfies (SH1)–(SH5). The $\bar{\phi}$-monomorphism $\psi : \mathcal{X}_{IL} \rightarrow \mathcal{X}_{BML}$ is the identity (using (M2)). The existence of ζ_2 can be proven analogously. ■

Definition 18 (Term-Rewrite-Systems for Back-Ends) Let $ML = merge(IL, BML)$ be the merged language. A *back-end term-rewrite rule* is a triple $rule = (t \rightarrow X; \{m_1; \ldots; m_n\})$ where $t \in T(\Upsilon_{IL}, V)$, $m_i \in T(\Sigma_{BML}, V)$ where each m_i has the form $m_i = f(t_1, \ldots, t_k)$ for an $f \in IM$, $t_i \in V \cup BITS^*$, and $X \in V_{ML} \cup \{\bullet\}$. The variables are called the *non-terminals* of the rule. A *term-rewrite system for back-ends* (TRSBE) is a set of back-end term rewrite rules.
Let $\pi \in ML$ be a program. *rule* is *applicable to an instruction instr* $\in INSTR_\pi$ iff there is an occurence o and a matching $\sigma : V \rightarrow T(\Sigma_{ML}, V)$ with $\sigma(instr[o]) = t'$ and for every v with $\sigma(v) \neq v$ there is an $a \in BITS^m$ such that $\sigma(v) = Reg(a)$ for an $a \in BITS^m$. If $X = \bullet$, then *rule* is only applicable if $o = \varepsilon$.
A *register assignment for the application of rule on instr at occurence o* is a substitution $\sigma_{\pi,instr,o}$ such that for $v = X$ and every v occurring in $m_1, \ldots, m_n$ which does not occur in t there is an $a \in BITS^m$ such that $\sigma_{\pi,instr,o}(v) = Reg(a)$. The *application of rule to an instruction instr* $\in INSTR_\pi$ *at occurence o* yields a program $\pi' \in ML$ (denoted $\pi \rhd \pi'$), where *instr* is replaced by the sequence of instructions $\sigma'(m_1); \ldots; \sigma'(m_n); instr[o/\sigma'(t)]$ where $instr[o]$ matches t with substitution σ, $\sigma' = \sigma \cup \sigma_{\pi,instr,o}$, and $\sigma_{\pi,instr,o}$ is a register assignment for the application of *rule* on *instr* at o. A TRSBE is *correct* iff π' is a correct compilation for every $\pi, \pi' \in BML$ such that $\pi \rhd^* \pi'$. ■

Remark: We allowed only bit sequences as arguments of machine instructions. If σ' is applied to a machine instruction, then a is substituted instead of $Reg(a)$. The precise formalization is left to the reader. As usual, $\rhd^*$ is the reflexive, transitive closure of $\rhd$. The code selection must find for any program π of the intermediate language IL a program $\pi' \in IL'$ such that $\pi \rhd^* \pi'$, and π' is a correct compilation of π. ■

Example 7 The following rules specify a small part of the compilation from BB into BB_α (cf. appendix A):

$$intassign(L,R) \to \bullet;\ \{STQ(R,0,L)\} \tag{1}$$
$$intassign(local(intconst_{i16}),R) \to \bullet;\ \{STQ(R,i16,1)\} \tag{2}$$
$$intadd(X,Y) \to Z;\ \{ADD(X,Y,Z,Q)\} \tag{3}$$
$$local(intconst_{i16}) \to X;\ \{LDQ(1,i16,X)\} \tag{4}$$
$$intadd(X,intconst(intconst_{i16})) \to Y;\ \{ADDI(X,i16,Y,Q)\} \tag{5}$$
$$intconst_{i32} \to X;\ \left\{\begin{array}{l} LDA(T1, i32.\mathsf{L}, 31) \\ ZBI(T1, \#11111100_2, T1) \\ LDAH(X, i32.\mathsf{H}, \mathsf{T1}) \end{array}\right\} \tag{6}$$
$$cont(X) \to Y;\ \{LDQ(X,0,Y)\} \tag{7}$$
$$content(local(intconst_{i16})) \to X;\ \{LDQ(1,i16,X)\} \tag{8}$$

If ik occurs in a rule, this is an abbreviation for 2^k rules: ik stands for any integer $i \in \{-2^k, \ldots, 2^k-1\}$. Register $R31$ is always zero. Rule (6) is necessary, since 32-bit integers cannot occur as operands of DEC-Alpha machine instructions. $i32$ denotes a 32-bit integer, $i32.\mathsf{L}$ denotes the lower two bytes of $i32$, and $i32.\mathsf{H}$ denotes the upper two bytes of $i32$. Table 4 shows the sequence of applications of rules and the register assignments for producing code of the statement $V := V+1$

$$intassign(local(intconst_8), intadd(cont(local(intconst_8)), intconst_1)).$$

Observe, that register $R1$ is preassigned and stores the current address defined by *local*. Observe, that instead of applying rule (8) we could also apply rules

Step	Program and Registers	Rule
0	$intassign(local(intconst_8), intadd(cont(local(intconst_8)), intconst_1))$	
1	$LDQ(1,8,3);\ intassign(local(intconst_8), intadd(Reg(3), intconst_1))$ $\sigma(X) = Reg(3)$	(8)
2	$LDQ(1,8,3);\ ADDI(3,1,3,Q);\ intassign(local(intconst_8), Reg(3))$ $\sigma(X) = Reg(3), \sigma(Y) = Reg(3)$	(5)
3	$LDQ(1,8,3);\ ADDI(3,1,3,Q);\ STQ(3,8,1)$ $\sigma(R) = Reg(3)$	(2)

Table 4: Term-Rewrite Based Compilation

(4) and (7) to load the address in a separate step. Also we could apply rules (6) and (3), instead of rule (5). It is easy to see that the code produced by the application of these more simple rules is worse than the code in Table 4. ■

Remark: In practice, it is possible to assign costs to each term-rewrite rule and to determine the cost optimal application of rules [Emmelmann 1992]. This requires a planning phase which covers the subterm with the rules to be applied. Figure 14 shows the result of the planning phase for the term and the sequence of rules of example 7. The term is visualized as a tree. In the second phase these

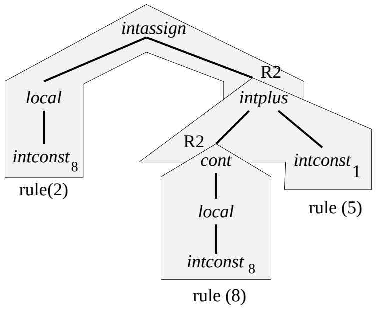

Figure 14: Planning of Code Selection by Term-Rewriting

rules are actually applied by the mechanism described above. We did not yet say anything about the concrete choice of registers.
[Emmelmann 1992] assumes that there are infinitely many registers available. After generation of code, these registers are assigned to the available registers. In contrast, we integrate register assignment with the planning phase. Further details are discussed in Subsection 4.4. ■
Since code selection can be viewed as a rewriting system $\rhd^*$ defined by a TRSBE, it is possible to define properties such as *confluence of a TRSBE* and a *Noetherian TRSBE*. Our TRSBE are always Noetherian if term-rewrite rules $X \to Y\ \{m_1, \ldots, m_k\}$, where X and Y are non-terminals, are excluded or ordered, respectively. Example 7 shows that $\rhd^*$ is usually not confluent.
It is not hard to see that the correctness of TRSBE depends on the register assignment:

Example 8 Consider the instruction

$$intassign(local(intconst_8), intadd(cont(local(intconst_8)), cont(local(intconst_{16}))))$$

Table 5 shows a code-generation for the term-rewriting in example 7. We assume that the register $R1$ contains the base address of the local environment. It is easy to see, that this compilation is incorrect, if $[\![content(local \oplus_A 8)]\!]_q \neq [\![content(local \oplus_A 16)]\!]_q$ in the state q before the execution of the statement. However, if we would replace in step (2) the assignment $\sigma(X) = R2$ with the assignment $\sigma(X) = R3$, then the produced code would be correct. Therefore, the rules are correct in a sense to be explained later.

The reason, why the compilation of the instruction in example 8 fails is that we wrote a value in register $R2$ although the old value would be needed. The problem for determining an adequate σ is called the *register assignment problem.* The basic idea is now to add a planning and normalization which preassigns registers and rules to the programs such that it can be decided during application, whether the value contained in the register is needed later in the execution or not. Furthermore, if there are not sufficiently many registers, then a source-to-source transformation is applied for storing the values of expressions into an unused memory cell. Again, the compiler must provide enough information to

Step	Program and Registers	Rule
(0)	$intassign(local(intconst_8),$ $intadd(cont(local(intconst_8)), cont(local(intconst_{16}))))$	
(1)	$LDQ(1,8,3);$ $intassign(local(intconst_8), intadd(Reg(3), cont(local(intconst_1 6))))$ $\sigma(X) = Reg(3)$	(8)
(2)	$LDQ(1,8,3);\ LDQ(1,16,3);$ $intassign(local(intconst_8), intadd(Reg(3), Reg(3)))$ $\sigma(X) = Reg(3)$	(8)
(3)	$LDQ(1,8,3);\ LDQ(1,16,3);\ ADD(3,3,3,Q);$ $intassign(local(intconst_8), Reg(3))$ $\sigma(X) = \sigma(Y) = \sigma(Z) = Reg(3)$	(3)
(4)	$mathitLDQ(1,8,3);\ LDQ(1,16,3);\ ADD(3,3,3,Q);\ STQ(3,8,1)$ $\sigma(X) = R2$	(2)

Table 5: Generation of code with erroneous register assignment

ensure that this memory cell is really unused. For simplicity, we assume that Δ_{TL} has enough registers.

4.4 Planning Term-Rewriting

The planning annotates programs with register assignments and term-rewrite rules such that it can be decided whether the application of a term-rewrite rule is legal under the register assignment. The basic idea is then to apply a term-rewrite rule conditionally using the annotations, i.e. at each sub-term which is annotated with a term-rewrite rule, this rule is applied using the register assignment annotations, provided it is legal. For simplicity, we assume that each register used for the evaluation of expressions is read just once, after it is written (i.e. common subexpressions are not eliminated). At the end of this subsection, we sketch a more general register assignment.

For the rest of this subsection, we assume that *IL* is a basic block language with operational semantics $\mathbb{A}_{IL}$, *TL* is a typical machine language with operational semantics $\mathbb{A}_{TL}$, *BML* is the basic block machine language obtained from *IL* and *TL*, *ML* is the merge of the languages *IL* and *BML* (cf. Figure 13), and R is a TRSBE.

Notation: In this subsection, the index *ML* is omitted for the components of language *ML* (e.g. Σ is the signature of *ML*-programs.

Definition 19 (Annotations) A *rule annotation* is a partial mapping $rule : PROG \times LABEL \times \mathbb{N} \times \mathbb{N}^* \to R$. A rule annotation *rule* for program π is *correct* iff for all (i, l, o) with $rule(\pi, i, l, o) = t' \to X;\ \{m_1, \ldots, m_n\}$, the following conditions are satisfied:

(RA1) $instr = get\_instr(\pi, i, l)$ is defined and $instr[o]$ is well-defined.
(RA2) $o = \langle\rangle$ iff $X = \bullet$.
(RA3) There is a substitution σ such that $t' = \sigma(instr[o])$, $rule(\pi, i, l, o \circ o') \neq \bot$ for all $o' \in \mathbb{N}^*$ satisfying $\sigma(v) \neq v, Reg(a)$, and for every

$v \in V$ satisfying $\sigma(v) \neq v$ an every $o'' \in \mathbb{N}^*$, $o'' \neq \langle\rangle$, which is a proper prefix of o', it holds $rule(\pi, i, l, o \circ o'') \neq \bot$.

(RA4) For every $i \in \mathbb{N}$, $l \in LABEL_\pi$, $get\_instr(\pi, i, l) = instr \in INSTR_\pi$ implies that $rule(\pi, i, l, \langle\rangle) \neq \bot$.

A *register annotation* w.r.t. a *rule annotation* is a mapping $regassign : PROG \times LABEL \times \mathbb{N} \times \mathbb{N}^* \rightarrow SET(BIT^m)$. *regassign* is *correct* for program π iff for all (i, l, t) the following conditions are satisfied:

(RA5) $rule(\pi, i, l, o) \neq \bot$ implies $regassign(\pi, i, l, o) \neq \bot$.
(RA6) $|regassign(\pi, i, l, o)|$ is the number of non-terminals occurring in $rule(\pi, i, l, o)$ which do not occur in the left hand side of this rule.
(RA7) If t' is the left hand side of $rule(\pi, i, l, o)$ and $\sigma(t') = instr[o]$, then for all $o' \in \mathbb{N}^*$ satisfying $\sigma(v) = instr[o \circ o'] \neq Reg(a)$, there is a $k \in regassign(\pi, i, l, o)$.
(RA8) Let RR be the set of registers used in the macro definitions specified by (BM4). Then $regassign(\pi, i, l, o) \cap RR = \emptyset$. ■.

Remark: (RA3) and (RA4) states that each instruction is covered by the patterns corresponding to the left hand side of the rules, i.e. each leaf of a pattern corresponds to the root of another pattern. Register assignments are associated with rule (cf. (RA5)). (RA6) states that there are enough registers to assign in order to apply the corresponding rule. The conditions in (RA7) states that the register assignments corresponding to the leaves of a pattern ensure that there enough registers to store the value for each leaf. The requirement (RA8) states that registers used for storing global information (e.g. the dynamic constants *loc* and *glob* in the basic block language *BB*, cf. Appendix A.1). ■

There are algorithms which compute correct rule annotations [Emmelmann 1992] and register allocation algorithms which compute correct register annotations [Waite and Goos 1984, Section 10.2.1]. For this article, it is sufficient to know that there are such algorithms. During the application of rules of R, annotations are consumed.

A rule $t \rightarrow X; \{m_1, \ldots, m_n\}$ is *applicable* iff the following two conditions are satisfied:

(AP1) $rule(\pi, i, l, o) = t \rightarrow X; \{m_1, \ldots, m_n\}$, and
(AP2) Let $instr = get\_instr(\pi, i, l)$. For all $k \in regassign(\pi, i, l, o)$ it is $k \notin RR$, and for all $o' \in \mathbb{N}^*$ $instr[o'] = Reg(k)$ implies that o is a prefix of o'.

(AP1) states that only those rules are applied which are annotated. The consequence of (AP2) is that every register $k \in regassign$ must not occur in *instr* outside of $t = instr[o]$. This is due to the following Lemma, which can easily be proven by induction using (AP1) and (AP2):

Lemma 20 (Invariant on Registers) Let $\pi \in IL$ be an annotated program. Then for any program $\pi' \in ML$ such that $\pi \rhd \pi'$, the following condition holds:

(AP3) For any instruction $instr \in INSTR_{\pi'}$ which is not a *BML*-instruction, there is no $k \in BIT^m$ such that $Reg(k)$ occurs more than once in *instr*. Furthermore there is no $k \in RR$ such that $Reg(k)$ occurs in *instr*.

Proof: Initially π contains no registers at all. (AP3) is therefore satisfied trivially. Suppose π satisfies (AP3) and $\pi \rhd \pi'$. By Definition 18 no term-rewrite rule can be applied onto machine instructions. Therefore there is a rule applied at occurence o of a non-machine instruction $instr = get\_instr(\pi, i, l)$. Then $instr' = instr[o/Reg(k)]$ for a $k \in regassign(\pi, i, l, o)$. (AP2) and (AP3) together imply that there is no other subterm of $instr'$ equal to $Reg(k)$. ■.
Consequently, each value written into a register is just read once. Conditions (AP1) and (AP2) can be checked obviously at compile time. We therefore specialize $\rhd$ such that a rule is applied only if (AP1) and (AP2) is satisfied. If we would know that for every $\pi, \pi' \in ML$, $\pi \rhd^* \pi'$, that π' is a correct compilation of π, then a compiler must just find a derivation from a $\pi \in IL$ to a $\pi' \in BML$. Obviously, there is an algorithm finding such a derivation just using the annotated program. Therefore, the code selection has the architecture shown in Figure 15. Thus, for construction of correct code selection it remains to show that fire every $\pi \in IL, \pi' \in BML$ such $\pi \rhd^* \pi'$, π' is a correct compilation of π.

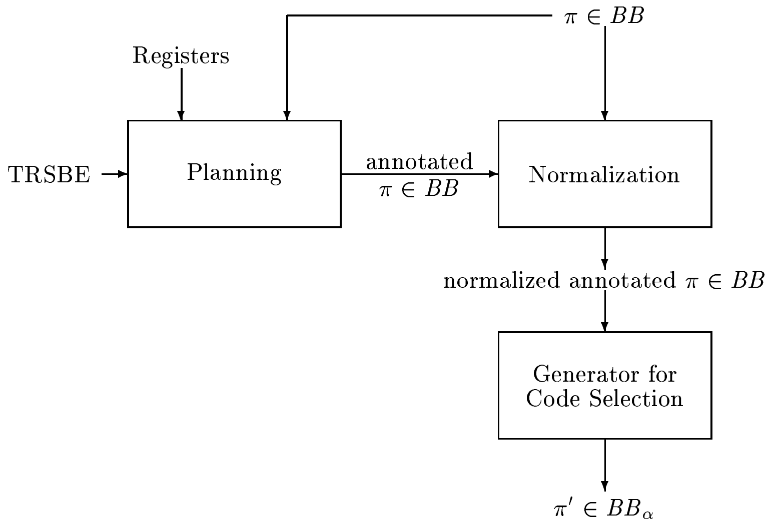

Figure 15: Architecture of Code Selection

Remark: If values of common subexpressions are stored in registers which are used later, then a new annotation is introduced referring to the register containing the value. In this case the requirement (AP2) is strengthen to assign no register whose value is still needed. Using the new annotation this can be computed again at compile time. ■

5 Correctness of Code Selection by Term Rewriting

In this section we show that it is sufficient to prove independently the local properties for each term-rewrite rule in a TRSBE in order to ensure the correctness of a TRSBE. Throughout this section we assume that *IL* is a basic block language with operational semantics $\mathbb{A}_{IL}$, *TL* is a typical machine language with operational semantics $\mathbb{A}_{TL}$, *BML* is the basic block machine language obtained from *IL* and *TL*, *ML* is the merge of the languages *IL* and *BML* (cf. Figure 13), and R is a TRSBE with the same assumptions as in subsection 4.4. In particular, we assume that any program $\pi \in ML$ is correctly annotated and $\triangleright$ is defined by the conditional application on conditions (AP1) and (AP2).
The approach is the following: First, we show that R is correct, if π' is a correct compilation of π for all $\pi, \pi' \in ML$ such that $\pi \triangleright \pi'$ by using Theorem 3. Then, we define a relation ρ and properties on the applied rule such that we can apply Theorem 5 to prove that π' is a correct compilation of π for all $\pi, \pi' \in ML$ such that $\pi \triangleright \pi'$. Finally, we give sufficient conditions for proving these properties of applied transformation rules. In particular, we show that these properties of single term-rewrite rules can be proven just using the macros and transition rules of *BML* and *IL*.

Theorem 8 Vertical Decomposition. If for all $\pi_1, \pi_2 \in ML$ such that $\pi_1 \triangleright \pi_2$, π_2 is a correct compilation of π_1, then π' is a correct compilation of π for every $\pi \in IL, \pi' \in BML$ satisfying $\pi \triangleright^* \pi'$.

Proof: Let $\pi \in IL$. Then also $\pi \in ML$. Let $\mathcal{A}_{\pi,IL}$ the ASM of π defined by $\mathbb{A}_{IL}$, and $\mathcal{A}_{\pi,ML}$ the ASM of π defined by $\mathbb{A}_{ML}$. Applying Theorem 3 inductively shows that $\pi' \in ML$ is a correct compilation of $\pi \in ML$, if $\pi \triangleright^* \pi'$. Thus, there is a relation $\rho \subseteq Q_{\pi,ML} \times Q_{\pi',ML}$ such that for every computation sequence $qq' \in \mathcal{A}_{\pi',ML}$ there is a computation sequence $qq \in \mathcal{A}_{\pi,ML}$ such that qq' ρ-simulates qq. Since there is an *BML*-semantics monomorphism $\zeta_2 : \mathbb{A}_{BML} \to \mathbb{A}_{ML}$ (cf. Theorem 7), there is an ASM-monomorphism $\xi_2 = (\bar{\phi}_2, \psi_2, \gamma_2) : \mathcal{A}_{\pi',BML} \to \mathcal{A}_{\pi',ML}$. From property (H3) follows immediately that for any computation sequence $\overline{qq}$ of $\mathbb{A}_{BML}$, there is a computation sequence qq' of $\mathbb{A}_{ML}$ such that $\overline{qq}$ γ_2^{-1}-simulates qq. Hence, for any computation sequence $\overline{qq}$ of $\mathbb{A}_{BML}$, there is a computation sequence $qq \in \mathcal{A}_{\pi,ML}$ such that qq' $(\gamma_2^{-1} \circ \rho)$-simulates qq.
Theorem 7 implies that there is also an *IL*-semantics monomorphism $\zeta_1 : \mathbb{A}_{ML} \to \mathbb{A}_{IL}$. Thus, there is an ASM-monomorphism $\xi_1 = (\bar{\phi}_1, \psi_1, \gamma_1) : \mathcal{A}_{\pi,ML} \to \mathcal{A}_{\pi,IL}$. (M4) ensures that $\gamma_1(I_{\pi,IL}) = I_{\pi,ML}$ and for any state $q \in \gamma(Q_{\pi,IL})$, $q \to_{\pi,ML} q'$ implies $q' \in \gamma(Q_{\pi,IL})$, i.e. γ_1 is bijective. Then we can argue as above to show that for any computation sequence $\overline{qq}$ of $\mathbb{A}_{BML}$ there is a computation sequence $qq \in \mathcal{A}_{\pi,IL}$ such that $\overline{qq}$ $\gamma_2^{-1} \circ \rho \circ \gamma_1$-simulates qq. ■
Thus, it is sufficient to show that π' is a correct compilation of π for all $\pi, \pi' \in ML$ such that $\pi \triangleright \pi'$. Observe that these compilations are source-to-source transformations. The basic idea is to define ρ adequately such that each single instructions which remains unchanged has the same effect in π and π'.
For the definition of ρ we have to know for each instruction in the program the registers containing a value which is required later. These informations can be computed easily when the register annotations are computed. Therefore, we assume an annotation $used : PROG \times LABEL \times \mathbb{N} \to SET(BIT^m)$. $r \in used(\pi, i, l)$ iff r contains a value that must not be destroyed. This annotation is updated

when applying term-rewrite rules. In our case, initially $used(\pi, i, l)$ is the set of registers used to represent $f \in \Delta_{IL} \setminus \Delta_{ML}$. When applying a term-rewrite rule $t \to X\{m_1, \ldots, m_n\}$ at an instruction *instr*, then the annotations of any unchanged instructions remain unchanged and the annotations for the new instructions is a simple live analysis for basic blocks (cf. [Waite and Goos 1984, Chapter 13.3]) starting from the set *used* of the instruction after *instr*.
Since we have now formalized the notion whether a register contains at a certain instruction a value which is needed later, we can define ρ.

Definition 21 **(ρ)** Let $\pi, \pi' \in ML$ where π' is obtained from π by applying $t \to X; \{m_1, \ldots, m_n\}$ at $instr = get\_instr(\pi, i, l)$ for a $i \in \mathbb{N}$, $l \in LABELS$. Furthermore, let $\mathcal{A}_\pi$ and $\mathcal{A}_{\pi'}$ be the ASMs defined by $\mathbb{A}_{ML}$. Then we define $\rho \subset Q_\pi \times Q_{\pi'}$ iff the following conditions are satisfied:

(ρ1) For all q such that $[\![BP]\!]_q = l' \neq l$: $[\![f]\!]_q = [\![f]\!]_{q'}$ for all $f \in \Delta_{ML} \setminus \{reg\}$ and $[\![reg(k)]\!]_q = [\![reg(k)]\!]_{q'}$ for all $k \in used(\pi, i, l)$.

(ρ2) The same properties defined by (ρ1) are also satisfied for all q with $[\![BP]\!]_q = l$ and $[\![PC]\!]_q \leq i$.

(ρ3) For all q such that $[\![BP]\!]_q = l$ and $[\![PC]\!] > i$: $[\![f]\!]_q = [\![f]\!]_{q'}$ for all $f \in \Delta_{ML} \setminus \{PC, reg\}$,

$$[\![PC]\!]_{q'} = \begin{cases} [\![PC]\!]_q + n & \text{if } X \neq \bullet \\ [\![PC]\!]_q + n - 1 & \text{if } X = \bullet \end{cases},$$

and $[\![reg(k)]\!]_q = [\![reg(k)]\!]_{q'}$ for all $k \in used(\pi, i, l)$. ∎

We now prove the precondition of Theorem 8:

Theorem 9 Horizontal Decomposition. Let $\pi, \pi' \in ML$ where π' is obtained from π by applying $t \to X; \{m_1, \ldots, m_n\}$ at $instr = get\_instr(\pi, i, l)$ for a $i \in \mathbb{N}$, $l \in LABELS$. Furthermore, let $\mathcal{A}_\pi$ and $\mathcal{A}_{\pi'}$ be the ASMs defined by $\mathbb{A}_{ML}$. Suppose that the following two conditions are satisfied:

(LC1) If $X \neq \bullet$, then for all $(q, q') \in \rho$ such that $[\![BP]\!]_q = l$ and $[\![PC]\!]_q = i$, and any sequence $q'_1, \ldots, q'_n, q'_{n+1} \in Q_{\pi_2}$ satisfying $q' \to_{\pi'} q'_1$ and $q'_i \to_{\pi'} q'_{i+1}$ for all $i = 1, \ldots, n$, then there is a state $\hat{q} \in Q_\pi$ such that $q \to_\pi \hat{q}$ and $(q, \hat{q}) \in \rho$.

(LC2) If $X \neq \bullet$, then for all $(q, q') \in \rho$ such that $[\![BP]\!]_q = l$ and $[\![PC]\!]_q = i$, and any sequence $q'_1, \ldots, q'_n, q'_n \in Q_{\pi_2}$ satisfying $q' \to_{\pi'} q'_1$ and $q'_i \to_{\pi'} q'_{i+1}$ for all $i = 1, \ldots, n-1$, then there is a state $\hat{q} \in Q_\pi$ such that $q \to_\pi \hat{q}$ and $(q, \hat{q}) \in \rho$.

Then, π' is a correct compilation of π.

Proof: We show that for any $(q, q') \in \rho$ where $[\![BP]\!]_q \neq l$ or $[\![PC]\!]_q \neq i$ and all states $q'' \in Q_{\pi'}$ such that $q' \to_{\pi_2} q''$, there is a state $\hat{q} \in Q_\pi$ satisfying $q \to_{\pi_1} \hat{q}$ and $(\hat{q}, q'') \in \rho$. If this is satisfied, we can use Theorem 5 to conclude together with (LC1), (LC2) that π' is a correct compilation of π. Let $instr = [\![IP]\!]_q$. Using (ρ1)–(ρ3) it is easy to see that also $[\![IP]\!]_{q'} = instr$. Let *trans* be the transition rule used to obtain q''. Then, *trans* can also be applied at q to obtain $\hat{q}$. Obviously, $q' \to_{\pi_2} q''$ executes an update $f(t_1, \ldots, t_n) := t_{n+1}$ iff $q \to_{\pi_1} \hat{q}$ executes the update $f(t_1, \ldots, t_n) := t_{n+1}$. The executed updates are equal iff

$[\![t_i]\!]_q = [\![t_i]\!]_{q'}$ for $i = 1, \ldots, n$ (except the update $pc := pc + 1$ for the case described by ($\rho 3$)). The latter can be proven by a simple structural induction on the terms $t_i \in T(\Xi \cup \Delta)$ using the fact that if t_i contains a subterm $reg(k)$ then $k \in used(\pi, l, i-1)$. ■

Figure 16 visualizes (LC1), (LC2) and the case described in the proof of Theorem 9.

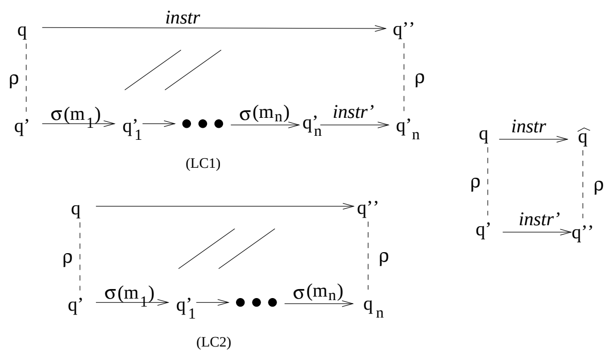

Figure 16: Visualization of (LC1), (LC2) and the Proof of Theorem 9

It follows immediately the

Corollary 22 Suppose that for all $\pi, \pi' \in ML$ where π' is obtained from π by applying $t \to X; \{m_1, \ldots, m_n\}$ at $instr = get\_instr(\pi, i, l)$ for a $i \in \mathbb{N}$, $l \in LABELS$, (LC1) and (LC2) is satisfied. Then for all $\pi_1 \in IL$, $\pi_2 \in BML$ such that $\pi_1 \rhd^* \pi_2$, π_2 is a correct compilation of π_1.

It is obvious that Definition 21 contains the minimal requirements such that Theorem 5 can be proven. It is natural to ask why it is not required that $(q, q') \in \rho$ implies $[\![reg(k)]\!]_q = [\![reg(k)]\!]_{q'}$ for every $k \in BITS^m$? The reason is that (LC1) and (LC2) need not to be satisfied. The machine instructions $\sigma(m_i)$ may write a value into register k not used by $instr$ but just by $instr'$ (e.g. the register used for X). Then $[\![reg(k)]\!]_{\hat{q}} \neq [\![reg(k)]\!]_{q_{n+1}}$ is possible. These are precisely such registers where Definition 21 does not require equality.

We finish the section by showing how to prove (LC1) and (LC2). First, we reduce (LC1) to expression evaluation.

Lemma 23 Let $\pi, \pi' \in ML$ where π' is obtained from π by applying $t \to X; \{m_1, \ldots, m_n\}$ on $instr = get\_instr(\pi, i, l)$ at occurence o for a $i \in \mathbb{N}$, $l \in LABELS$. Furthermore, let $\mathcal{A}_\pi$ and $\mathcal{A}_{\pi'}$ be the ASMs defined by $\mathbb{A}_{ML}$. Suppose $X \neq \bullet$ and let the states $q, q', q'_1, \ldots, q'_{n+1}$ be defined as by (LC1). Let o σ be the substitution used in the application of that rule. Furthermore, assume $\sigma(X) = Reg(k)$. If $[\![NF(eval(instr[o]))]\!]_q = [\![reg(k)]\!]_{q_n}$, then (LC1) is satisfied.

Proof: It is sufficient to show that for every $o \neq \langle\rangle$ where $e = instr[o]$ is an expression, the following two properties hold.

(i) $[\![NF(eval(e))]\!]_{q'_n} = [\![NF(eval(e)]\!]_q$ if $Reg(k)$ does not occur in e, and
(ii) $[\![NF(eval(e[o/Reg(k)]))]\!]_{q'_n} = [\![NF(eval(e)]\!]_q$ if $Reg(k)$ occurs e.

Then we can argue as in the proof of Theorem 5 that there is a $\hat{q} \in Q_\pi$ such that $q \to_\pi \hat{q}$ performs the same updates as $q'_n \to q'_{n+1}$. Since $m_1, \ldots, m_n$ contains no registers $r \notin used(\pi, i, l)$, and every $\sigma(m_i)$ only changes contents of registers and pc, it follows $[\![f]\!]_{q'_n} = [\![f]\!]_q$ for every $f \in \Delta \setminus \{reg, PC\}$ and $[\![reg(r)]\!]_{q'_n} = [\![reg(r)]\!]_q$ for all $r \in used(\pi', i+n, l) \setminus \{k\}$. Using these properties and $[\![NF(eval(instr[o]))]\!]_q = [\![reg(k)]\!]_{q_n}$, it is now a simple structural induction to prove every subterm of $instr'$ is evaluated to the same value. ■
From this Lemma and Corollary 22 follows immediately the

Corollary 24 Suppose that for all $\pi, \pi' \in ML$ where π' is obtained from π by applying $t \to X; \{m_1, \ldots, m_n\}$ at $instr = get\_instr(\pi, i, l)$ for a $i \in \mathbb{N}$, $l \in LABELS$, (LC2) and the following condition holds:

(LC3) If $X \neq \bullet$, then for all states $(q, q') \in \rho$ such that $[\![BP]\!]_q = l$ and $[\![PC]\!]_q = i$, and any sequence $q'_1, \ldots, q'_n, q'_{n+1} \in Q_{\pi_2}$ satisfying $q' \to_{\pi'} q'_1$ and $q'_i \to_{\pi'} q'_{i+1}$ for all $i = 1, \ldots, n$, it is $[\![NF(eval(t'))]\!]_q = [\![reg(k)]\!]_{q'_n}$, where $\sigma(X) = Reg(k)$ is the register assigned to X when applying the term-rewrite rule

Then π_2 is a correct compilation of π_1 for all programs $\pi_1 \in IL, \pi_2 \in BML$ such that $\pi_1 \rhd^* \pi_2$. ■

Conditions (LC2) and (LC3) are called the *local correctness conditions* of rule $t \to X; \{m_1, \ldots, m_n\}$. The next section shows how these local correctness conditions can be proven.
It is remarkable that Corollary 24 is just based on the general requirements defined by basic block languages and typical machine languages. The other assumptions (A1)–(A6), the property of assigning exclusively registers, and that expressions have no side effects can be removed, but would complicate considerably the proofs. Removing (A1)–(A6) would lead to a more complex monomorphism $\zeta_1 : \mathbb{A}_{IL} \to \mathbb{A}_{ML}$ is more complicated. Adding memory locations is not difficult: it is just an extension of the annotation (although the normalization could be defined such that the approach described in this section is always applicable). If expressions can also have side-effects then a combination of (LC2) and (LC3) is necessary.

6 Correctness of Term Rewrite Rules

In Section 5 we reduced the correctness of a TRSBE T to proving the local correctness of T (cf. Corollary 24). However (LC2) and (LC3) quantify over all states. This suggests to execute the state transitions symbolically using the rules in $\mathbb{A}_{ML}$. In particular:

1. If the rule is $t \to X\ \{m_1; \ldots; m_n\}$, the proof proceeds by the following steps: First, evaluate e symbolically, i.e. compute $NF(eval(t))$. Then the updates by the transition rules $m_i \rightsquigarrow rhs$ are executed symbolically and normalized for $i = 1, \ldots, n$ in that order. Finally $reg(k)$ and $NF(eval(t))$ are compared. If they are equal then (LC3) is satisfied.

2. If the rule is $t \to \bullet \ \{m_1; \ldots; m_n\}$ the proof proceeds by the following steps: First, the updates of the rule $t \rightsquigarrow \mathit{rhs}$ are performed symbolically. Then the updates by the transition rules $m_i \rightsquigarrow \mathit{rhs}$ are executed symbolically and normalized for $i = 1, \ldots, n$ in that order. Then, we have to compare all dynamic functions updated by one of the rules.

These two proof strategies are mechanized using the proof checker PVS [Dold and Gaul 1996]. They are justified by the way how ASMs are obtained from the operational semantics: The transition rules for the instructions of the concrete program is obtained by substituting the variables in the corresponding rules in *Trans* and simplifying them using *Macros*:
Consider for example a program $\pi \in \mathit{ML}$ and a instruction $\mathit{instr} = \mathit{get\_instr}(\pi, i, l)$, $i \in \mathbb{N}, l \in \mathit{LABELS}$. Suppose $f(x_1, \ldots, x_n) \rightsquigarrow \mathit{rhs} \in \mathit{Trans}$ such that there is a substitution σ with $\sigma(f(x_1, \ldots, x_n)) = \mathit{instr}$. The, a symbolic update $\mathit{lhs} := \mathit{rhs}$ is executed by the above approach iff the transition rule for instr executes the update $\mathit{NF}(\sigma(\mathit{lhs} := \mathit{rhs}))$. Similarly, we have for any subexpression e' of t $\mathit{NF}(\mathit{eval}(e')) = s$ iff $[\![\mathit{NF}(\mathit{eval}(\sigma(e)))]\!]_q = [\![\mathit{NF}(\sigma(s))]\!]_q$ for all $q \in Q_\pi$.
We show now three typical local correctness proofs according to the above strategies for the languages defined in Appendix A. ML denotes the mnerge of the languages BB and BB_α. For other proofs, we refer to [Dold and Gaul 1996].

Lemma 25 (Local Correctness of Rule 3) Let $\pi, \pi' \in \mathit{ML}$ be arbitrary programs with $\pi \rhd \pi'$, $\mathcal{A}_\pi$ and $\mathcal{A}_{\pi'}$ their ASMs in $\mathbb{A}_{ML}$, $q \in Q_\pi$ a state with $[\![\mathit{IP}]\!]_q = \mathit{instr}$ where the rule

$$\mathit{intadd}(X, Y) \to Z \ \{\mathit{ADD}(X, Y, Z, Q)\}$$

is applied onto instr to obtain π from π', σ the corresponding substitution, $q' \in Q_{\pi'}$ be a state such $[\![\mathit{IP}]\!]_{q'} = \mathit{ADD}(\sigma(X), \sigma(Y), \sigma(Z), Q)$, and $q'' \in Q_{\pi'}$ be the state such that $q' \to_{\pi'} q''$. Then, for any $\rho \subset Q_\pi \times Q_{\pi'}$ satisfying Definition 21 $(q, q') \in \rho$ implies $[\![\mathit{eval}(\mathit{intadd}(\sigma(X), \sigma(Y)))]\!]_q = [\![\mathit{Reg}_{quad}(\sigma(Z))]\!]_{q''}$.

Proof: By the definition of *eval* (cf. Appendix A.1) we obtain

$$[\![\mathit{eval}(\mathit{intadd}(\sigma(X), \sigma(Y)))]\!]_q = [\![\mathit{Reg}_{quad}(\sigma(X))]\!]_q \oplus_I [\![\mathit{Reg}_{quad}(\sigma(Y))]\!]_q. \quad (9)$$

The execution of $\mathit{ADD}(\sigma(X), \sigma(Y), \sigma(Z), Q)$ performs the update
$\mathit{Reg}_{quad}(\sigma(Z)) := \mathit{Reg}_{quad}(\sigma(X)) \oplus_I \mathit{Reg}_{quad}(\sigma(Y))$ (cf. the rule `ADD` in Appendix A.2). Thus

$$[\![\mathit{Reg}_{quad}(\sigma(Z))]\!]_{q''} = [\![\mathit{Reg}_{quad}(\sigma(X))]\!]_{q'} \oplus_I [\![\mathit{Reg}_{quad}(\sigma(Y))]\!]_{q'}. \quad (10)$$

Since $(q, q') \in \rho$, $[\![\mathit{Reg}_{quad}(\sigma(X))]\!]_q = [\![\mathit{Reg}_{quad}(\sigma(X))]\!]_{q'}$ and $[\![\mathit{Reg}_{quad}(\sigma(Y))]\!]_q = [\![\mathit{Reg}_{quad}(\sigma(Y))]\!]_{q'}$. Thus, the right hand sides of (9) and (10) are equal and therefore

$$[\![\mathit{eval}(\mathit{intadd}(\sigma(X), \sigma(Y)))]\!]_q = [\![\mathit{Reg}_{quad}(\sigma(Z))]\!]_{q''}.$$

■

Rule 6 for loading 32-bit integer constants generates more than one machine instructions.

Lemma 26 (Local Correctness of Rule 6) Let $\pi, \pi \in ML$ be arbitrary programs with $\pi \rhd \pi'$, $\mathcal{A}_\pi$ and $\mathcal{A}_{\pi'}$ their ASMs in $\mathbb{A}_{ML}$, $q \in Q_\pi$ a state with $[\![IP]\!]_q = instr$ where rule 6 is applied onto *instr* to obtain π from π', σ be the corresponding substitution, $q' \in Q_{\pi'}$ be a state such $[\![IP]\!]_{q'} = LDA(\sigma(T1), i32\langle 0:15\rangle, 31)$, and $q'' \in Q_{\pi'}$ be the state such that $q' \to_{\pi'} q_1 \to_{\pi'} q_2 \to_{\pi'} q''$. Then, for any $\rho \subset Q_\pi \times Q_{\pi'}$ satisfying Definition 21 $(q, q') \in \rho$ implies

$$[\![eval(intconst_{i32})]\!]_q = [\![Reg_{quad}]\!]_{q''}(\sigma(X))$$

.

Proof: The transition rules for machine instructions *LDA* (*load address*) and *ZBI* (*zero-bytes-immediate*) are defined in Appendix A.2. The proof uses the following definitions: $s_l = (i32)\langle 15\rangle$ and $s_h = (i32)\langle 31\rangle$
With these definitions, we obtain the following equalities using the macros in Appendix A.2:

$$Sext_{16}(i32\langle 0:15\rangle) = s_l^{48} \circ (i32)\langle 0:15\rangle \tag{11}$$
$$Sext_{16}(i32\langle 16:31\rangle) = s_h^{48} \circ (i32)\langle 16:31\rangle \tag{12}$$

and $ByteZap(x, o) = extract(x, 7, (o)\langle 7\rangle) \circ \cdots \circ extract(x, 0, (o)\langle 0\rangle)$, where

$$extract(x, i, b) = \begin{cases} 00000000 & \text{if } b = 1 \\ (x)\langle i \cdot 8 : i \cdot 8 + 7\rangle & \text{if } b = 0 \end{cases}$$

$$\text{E.g. } ByteZap(x, 11111100) = 0^{48} \circ (x)\langle 0:15\rangle. \tag{13}$$

Similarly, $LogShift_L(x, n) = (x)\langle 0 : 63 - n\rangle \circ 0^n$, e.g.

$$LogShift_L(Sext_{16}(i32\langle 16:31\rangle), 16) = s_h^{32} \circ (i32)\langle 16:31\rangle \circ 0^{16}. \tag{14}$$

By the definition of *eval* (cf. Appendix A.1), it is

$$[\![eval(intconst_x)]\!]_q = x \tag{15}$$

The transition rule for *LDA* (cf. Appendix A.2) implies that the following update is performed to obtain q_1: $Reg_{quad}(\sigma(T1)) := Reg_{quad}(31) \oplus_I Sext_{16}(i32\langle 0:15\rangle)$. With the above definitions, (11), and the fact that on the DEC-Alpha $[\![Reg_{quad}]\!]_{\hat{q}}(31) = 0$ for all states $\hat{q}$, we obtain:

$$[\![Reg_{quad}(\sigma(T_1))]\!]_{q_1} = s_l^{48} \circ (i32)\langle 0:15\rangle \tag{16}$$

Then the instruction $ZBI(\sigma(T1), \#11111100, \sigma(T1))$ is executed. The transition rule for *ZBI* shows, that the update $Reg_{quad}(\sigma(T1)) :=$
$ByteZap(Reg_{quad}(\sigma(T1)), 11111100)$ is performed. Thus, we obtain from (13) and (16)

$$[\![Reg_{quad}(\sigma(T_1))]\!]_{q_2} = 0^{48} \circ (i32)\langle 0:15\rangle. \tag{17}$$

q'' is then reached by executing $LDAH\ (X, i32\langle 16:31\rangle, T1)$. Hence, on q_2 the update $Reg_{quad}(\sigma(X)) := Reg_{quad}(\sigma(T1)) \oplus_I LogShift_L(Sext_{16}(i32\langle 16:31\rangle), 16)$ is performed. With (17) and (14) we obtain:

$$\begin{aligned}
[\![Reg_{quad}(\sigma(X))]\!]_{q''} &= (0^{48} \circ (i32)\langle 0:15\rangle) \oplus_I (s_h^{32} \circ (i32)\langle 16:31\rangle \circ 0^{16}) \\
&= (s_h^{32} \circ (i32)\langle 16:31\rangle \circ (i32)\langle 0:15\rangle) && (18) \\
&\qquad \text{(Definition of } \oplus_I) \\
&= i32 && (19)
\end{aligned}$$

■

Remark: In our first attempt rule 6 was designed erroneously. We forgot that the instruction LDA applies a sign extension onto the 16-bit integer operand. Appendix B shows the effect when we try to prove the faulty version of rule 6. ■

The above lemmas are proven according to the first strategy. We finish this section with proving the local correctness of rule 2 by the second strategy.

Lemma 27 (Local Correctness of Rule 2) Let $\pi, \pi' \in L$ be arbitrary programs with $\pi \rhd \pi'$, $\mathcal{A}_\pi$ and $\mathcal{A}_{\pi'}$ their ASMs in $\mathbb{A}_{ML}$, $q \in Q_\pi$ a state with $[\![IP]\!]_q = intassign(local(intconst_{i16}), Reg(i))$, $\hat{q}$ be the state such that $q \to_\pi \hat{q}$, $q' \in Q_{\pi'}$ be a state such $[\![IP]\!]_{q'} = STQ\ (i, i16, 1)$, and $q'' \in Q_{\pi'}$ be the state such that $q' \to_{\pi'} q''$. Then, for any $\rho \subset Q_\pi \times Q_{\pi'}$ satisfying Definition 21 $(q, q') \in \rho$ implies $[\![content]\!]_{\hat{q}} = [\![content]\!]_{q''}$.

Proof: The rule for $intassign$ (cf. Appendix A.1) shows, that the transition to $\hat{q}$ performs the update $content(eval(local(intconst_{i16})) := eval(Reg(i))$. By the definition of BB_α (cf. Subsection A.3), $eval(Reg(i)) = reg_{quad}(Reg(i))$ and by the definition of $eval$, $eval(local(intconst_{i16})) = loc \oplus_A i16$. Since on BB_α, we have chosen $loc \hat{=} reg_{quad}(1)$, we have

$$[\![content]\!]_{\hat{q}}(a) = \begin{cases} [\![reg_{quad}(i)]\!]_q & \text{if } a = [\![reg_{quad}(1)]\!]_q \oplus_A i16 \\ [\![content(a)]\!]_q & \text{otherwise} \end{cases} \qquad (20)$$

By the rule for STQ (cf. Appendix A.2), the following update is performed on q':

$$content(reg_{quad}(1) \oplus_I Sext_{16}(i16)) := reg_{quad}(i)$$

Since $\oplus_I = \oplus_A$ and $Sext_{16}(i16) = i16$ algebraically, it holds

$$[\![content]\!]_{q''}(a) = \begin{cases} [\![reg_{quad}(i)]\!]_{q'} & \text{if } a = [\![reg_{quad}(1)]\!]_{q'} \oplus_A i16 \\ [\![content(a)]\!]_{q'} & \text{otherwise} \end{cases} \qquad (21)$$

If $(q, q') \in \rho$, then $[\![reg_{quad}(1)]\!]_q = [\![reg_{quad}(1)]\!]_{q'}$, $[\![reg_{quad}(i)]\!]_q = [\![reg_{quad}(i)]\!]_{q'}$, and $[\![content]\!]_q = [\![content]\!]_{q'}$. Hence, the right hand sides of (20) and (21) are equal. Thus, $[\![content]\!]_{\hat{q}} = [\![content]\!]_{q''}$. ■

7 Conclusions

In this article we showed how to construct correct compiler back-ends which transform intermediate languages (basic block graphs) into binary machine code with BURS. First, the problem was decomposed by introducing intermediate languages based on Theorem 3. The code generation works in two phases: First the basic block structure is kept while intermediate language instructions are transformed into machine instructions (code selection). Second, the basic block graphs with machine instructions are mapped into the memory of the target machine (code linearization). The focus of this article was on the construction of a correct code selection. The approach is based on a well-known code generation technology used in practice, the term-rewrite systems. The latter are specifications for code selections. A correct generator which performs term-rewriting can be used for obtaining a correct code selection, provided the specification used for generation was correct.
We reduced the correctness of term-rewriting systems T to proving independently for each rule of T a local correctness condition (Corollary 24) by conditionally applying the rules. The condition is a requirement on register assignment. In section 6 we showed two simple, mechanizable proof strategies for proving the local correctness.
Except of the local correctness of term-rewrite rules, none of the proofs in this article made specific assumptions on the instruction set of the intermediate language and target machine. Hence, these proofs need not be redone if a new back-end is designed. We showed that a generator can be parametrized with a term-rewrite system, the intermediate language, the target language, and the register assignment algorithms. Therefore, if such correct generators and register assignments are available, only the local correctness of the term-rewrite rules is required for construction of correct code selection. Since the correctness of register assignments is checked when term-rewrite-rules are applied, register assignment algorithms can be used *without proving their correctness*. Therefore, we can apply different register assignment algorithms until the correctness condition is satisfied. For completeness it is just necessary to apply one verified register assignment algorithm. This idea is similar to the idea of program checking [Blum and Kannan 1995]. In summary, for the construction of a correct code selection, it is sufficient to prove the local correctness of the term-rewriting system specifying the code selection.
First experiments show that the quality of the binary machine code generated by our correct compiler back-ends is orders of magnitudes faster than code generated by correct compilers constructed by other approaches [Palsberg 1992, Diehl 1996]. [Diehl 1996] has the best results so far. Table 6 shows the comparison between our approach, the approach in [Diehl 1996] (SIMP), and a standard unverified C-compiler. Loop is a program that initializes a variable with a positive integer and decrements this integer by one until the content of this variable is zero, Sieve implements the sieve of Eratosthenes.
Our correct compiler back-end is the first work on the construction of correct compilers which produces binary machine code whose performance is on the same order of magnitude as unverified standard C-compilers. For improving the code performance, new term-rewrite rules may be added. For keeping the correctness of code selection it is sufficient to prove the local correctness of the new rules. Thus, our approach allows incremental improvement of the code selection.

		DEC-Alpha				Intel-Pentium		SIMP
		Verifix		C-Compiler		C-Compiler		AM in C
	Iterations	non-opt	opt	non-opt	opt	non-opt	opt	min
Loop	10000	0.57ms	0.57ms	0.35ms	0.31ms	0.62ms	0.50ms	5.0s
	100M	5.72s	5.70s	3.49s	3.05s	6.12s	5.04s	13h53m*
Sieve	1	1.63ms	1.23ms	0.82ms	0.56ms	1.02ms	0.89ms	4.00s
	10000	16.35s	12.26s	8.25s	5.65s	10.23s	8.94s	11h6m*

DEC-Alpha: DEC-AXP(233MHz), OSF1, CC: DEC(V4.2)
Intel-Pentium: Pentium(133MHz), Linux, CC: GNU(V2.7.0)
SIMP: Pentium, execution times from [Diehl 1996], abstract machine implemented in C
Iterations: Loop: loop iterations, Sieve: searching the primes less than thousand, n times repeated
SIMP: line 1 from [Diehl 1996], line 2 extrapolation(*) on repeated iterations
Optimization (opt): Verifix: Peephole, C: Option -O4, SIMP: minimal execution times

Table 6: Comparison of the Performance of the Machine Code generated by Correct Compilers

Our vision is that correct compilers can be constructed by well-known compilation techniques, and if a library of correct data structures, algorithms, and generators is provided, then for the correctness of any transformation of one intermediate language to another, it is sufficient to prove local correctness properties of transformation rules similar to those of term-rewrite rules. The above performance results show that this approach seems feasible to construct realistic correct compilers compiling programs of real-life programming languages into binary machine code of real processors, and produce efficient code.

Acknowledgements: This work is supported by the *Deutsche Forschungsgemeinschaft* project Go 323/3-1 *Verifix* (Construction of Correct Compilers). We are grateful to our colleagues in *Verifix*. Especially, we thank Axel Dold for formalizing and checking the local correctness proofs in this article with the proof checker PVS. We would like to thank the anonymous referees for reading it very carefully. Their comments helped to improve our article drastically. Our colleagues Uwe Assmann, Axel Dold, Sabine Glesner, Gerhard Goos and Martin Trapp carefully read a preliminary version of this article and suggested us considerable improvements.

A Abstract Machines for the Languages

Subsection A.1 introduce abstract state machines for basic block graphs. Subsection A.2 introduces the abstract state machines for the DEC-Alpha processor family. It is not our purpose to show how these descriptions can be obtained from informal language definitions. We refer to [Gurevich and Huggins 1993, Wallace 1995]. Subsection A.3 describes the DEC-Alpha basic block graphs obtained after code selection.

A.1 Basic Block Graphs *BB*

A *BB*-program is given by a set of basic blocks where each block consists of a sequence of instructions where the last one in a block is a jump or stop. *INSTR* denotes the universe of instructions. The data types are the type of 64-bit integers INT, the double precision floating point numbers FLT, the booleans $BOOL$, and the addresses $ADDR$ on the target machine. $VALUE$ denotes the union of all those universes. Expressions are defined on these types and include integer and floating point expressions, boolean and address expressions ($INTEXPR$, $FLTEXPR$, $ADDREXPR$, $BOOLEXPR$), $EXPR$ is the sort which is a union of these expressions ($EXPR = INTEXPR \cup FLTEXPR \cup \ldots$). Expressions are evaluated by $eval : EXPR \to VALUE$ which is defined recursively over its structure. The semantics is parameterized with the data types and the basic operations of the target machine. The ASMs have the following dynamic functions: an program counter ($PC : INSTR$), a basic block pointer (BP), a pointer to the local environment ($loc : ADDR$), a pointer to the global environment ($glob : ADDR$), a history $hist \hat{=} LABEL \times \mathbb{N}^*$ which contains the stack of procedure calls not yet completed, and the memory which is accessed with dynamic $content : ADDR \to CELL$. The access to the memory is relative to loc or $glob$. In the program this is denoted by $\underline{local(i)}$ and $\overline{\underline{global(i)}}$, respectively. For accessing larger values, we use the macros $content_i$, $\overline{content_i}$ shown in Figure 11.

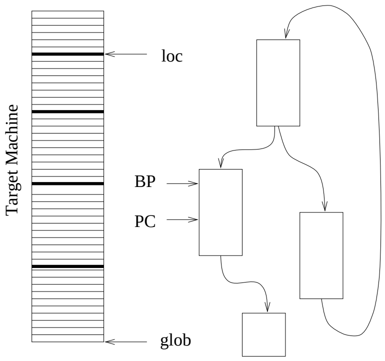

Figure 17: Basic block graphs

A.1.1 The language specification

The sorts of BB are

$$S_{BB} = \{ \; LABEL, INTEXPR, FLTEXPR, BOOLEXPR, ADDREXPR, \\ INSTR, JUMP, BLOCK, PROG \; \}$$

The signature of programs is

$$\begin{aligned} \Sigma_{BB} = \{ \; & intadd : INTEXPR \times INTEXPR \to INTEXPR \\ & fltadd : FLTEXPR \times FLTEXPR \to FLTEXPR \\ & intsub : INTEXPR \times INTEXPR \to INTEXPR \\ & \ldots \\ & int2flt : INTEXPR \to FLTEXPR \\ & intequal : INTEXPR \times INTEXPR \to BOOLEXPR \\ & intgreater : INTEXPR \times INTEXPR \to BOOLEXPR \\ & \ldots \\ & fltequal : INTEXPR \times INTEXPR \to BOOLEXPR \\ & \ldots \end{aligned}$$

$$
\begin{aligned}
&\mathit{booland} : BOOLEXPR \times BOOLEXPR \to BOOLEXPR\\
&\ldots\\
&\mathit{readint} : INTEXPR \to INSTR\\
&\mathit{writeint} : INTEXPR \times INTEXPR \to INSTR\\
&\mathit{intassign} : ADDREXPR \times INTEXPR \to INSTR\\
&\mathit{fltassign} : ADDREXPR \times FLTEXPR \to INSTR\\
&\ldots\\
&\mathit{condjump} : BOOLEXPR \times LABEL \times LABEL \to JUMP\\
&\mathit{jump} : LABEL \to JUMP\\
&\mathit{call} : LABEL \times \mathbb{N} \to INSTR\\
&\mathit{return} : \mathbb{N} \to INSTR\\
&\mathit{stop} : JUMP\\
&\mathit{local} : INTEXPR \to ADDREXPR\\
&\mathit{global} : INTEXPR \to ADDREXPR\\
&\mathit{cont} : ADDREXPR \to EXPR\\
&\mathit{intconst}_i : INTEXPR(-2^{31} \leq i < 2^{31})\\
&\mathit{boolconst}_b : BOOLEXPR(b \in \{true, false\})\\
&\ldots\\
&\mathit{newblock} : LABEL \times INSTR^* \to BLOCK\\
&\mathit{makeprog} : LABEL \times BLOCK^* \to PROG \ \ \}
\end{aligned}
$$

The control flow is defined by

$$
\begin{aligned}
\Gamma_{BB} = \{\ &\mathit{start} : PROG \to LABEL\\
&\mathit{get\_instr} : \mathbb{N} \times BLOCK \to INSTR\\
&\mathit{get\_block} : LABEL \times PROG \to BLOCK\\
&\mathit{next} : \mathbb{N} \to \mathbb{N} \ \ \}
\end{aligned}
$$

The interpretation is defined in Figure 9.
The instruction pointer is defined by $IP \ = \ \mathit{get\_instr}(PC, \mathit{block}(BP, \mathit{prog}))$

A.1.2 Operational semantics

$\oplus_A$ is the add operation on addresses of the machine, which is in our case equivalent to $\oplus_I$. Instructions consist of assignment instructions for different kind of expressions, jumps and procedure calls.
The universes not defined in the language are

$$U = \{\ ADDR, INT, FLT, BOOL, CELL\ \}$$

The interpretation of these universes are 64-bit sequences, except $CELL$, which is an 8-bit sequence.

$$
\begin{aligned}
\Xi = \{\ &0_A, 1_A, \ldots\ : ADDR\\
&\ldots, -1_I, 0_I, 1_I, \ldots\ : INT\\
&\oplus_I : INT \times INT \to INT
\end{aligned}
$$

$\oplus_F : INT \times INT \to FLT$
$\oplus_A : ADDR \times INT \to ADDR$
$\ominus_I : INT \times INT \to INT$
$\ldots$
$=_I: INT \times INT \to BOOL$
$=_F: FLT \times FLT \to BOOL$
$\ldots$
$\wedge_B : BOOL \times BOOL \to BOOL$
$true : BOOL$
$false : BOOL \;\}$

The operations on the universes are defined as by the DEC-Alpha machine language. The constants represent bit sequences.
Dynamic functions:

$$\begin{aligned}\Delta_{BB} = \{ \; & PC : \mathbb{N} \quad (instruction\ counter)\\ & BP : LABEL \quad (basic\ block\ pointer)\\ & content : ADDR \to CELL \quad (the\ memory)\\ & loc : ADDR \quad (current\ address\ of\ local\ procedure\ variables)\\ & glob : ADDR \quad (address\ of\ global\ variables)\\ & inp : VALUE^* \quad (input\ stream)\\ & out : VALUE^* \quad (output\ stream)\\ & hist : (LABEL \times \mathbb{N})^* \;\; \}\end{aligned}$$

The macros needed for the evaluation of expressions (macros defining *content* are shown in Figure 11)

Θ_{BB}:

$$\begin{aligned} eval(intadd(e_1, e_2)) &\hat{=} eval(e_1) \oplus_I eval(e_2)\\ eval(fltadd(e_1, e_2)) &\hat{=} eval(e_1) \oplus_F eval(e_2)\\ eval(intsub(e_1, e_2)) &\hat{=} eval(e_1) \ominus_I eval(e_2)\\ &\ldots\\ eval(booland(e_1, e_2)) &\hat{=} eval(e_1) \wedge_B eval(e_2)\\ &\ldots\\ eval(intconst_c) &\hat{=} c_I\\ eval(intequal(e_1, e_2)) &\hat{=} eval(e_1) =_I eval(e_2)\\ &\ldots\\ eval(local(e) &\hat{=} loc \oplus_A eval(e)\\ eval(global(e) &\hat{=} glob \oplus_A eval(e)\\ eval(cont(i)) &\hat{=} content_8(eval(i))\end{aligned}$$

We finish the definition with some transition rules:

ASM1

if $IP = intassign(a, e)$ **then**
  $content_8(eval(a)) := eval(e)$;
  $PC := next(PC)$
endif

ASM2

if $IP = jump(b)$ **then**
  $BP := b$;
  $PC := 0$
endif

if $IP = condjump(e, b_1, b_2)$ **then**
  if $eval(e)$ **then** $BP := b_1$;
    $PC := 0$;
  else $BP := b_2$;
    $PC := 0$
  endif
endif

ASM3

if $IP = call(P, k)$ **then**
  $loc := loc \oplus_A k$;
  $hist := (BP, PC).hist$;
  $BP := P$;
  $PC := 0$
endif

if $IP = return(k)$ **then**
  $loc := loc \ominus_A k$;
  $hist := tail(hist)$;
  $BP := fst(head(hist))$;
  $PC := next(snd(head(hist)))$
endif

The initializations are:

$$
\begin{aligned}
BP &:= start \\
PC &:= 0 \\
loc &:= bot\_of\_stack \\
glob &:= bot\_of\_stack
\end{aligned}
$$

where bot_of_stack is an external constant defined by the operating system.

A.2 The Dec-Alpha Processor Family L_α

In this section we sketch the formal representation of the DEC-Alpha based on the informal specification in the manufacturer manual [Sites 1992]. The formalization shows parts of the derived language specification and the operational semantics. It includes the instruction set, addressing modes, register files and the memory, i.e. it models the programmer's view. We do not describe the complete instructions of the DEC-Alpha assembly language. We describe only those used in this article, more details can be found in [Gaul and Zimmermann 1995]. The addressable memory unit is a byte. In order to load and store quadwords – the usual integer type for DEC-Alpha architectures – or floats we introduce the function $content_8 : QUAD \rightarrow VALUE$ which loads and stores 8 bytes from/into memory. For example, fetching a quadword or float from memory is carried out by concatenating 8 subsequent bytes starting at the given address. Register are always accessed as full QUADs, that means there is no byte-access to registers.

A.2.1 The language specification

The sorts of programs are $ADDR$, $CELL$, $INSTR$, *and* $PROG$. The interpretation of $ADDR$ and $CELL$ is isomorphic to BIT^{64} and $BYTE \widehat{=} BIT^8$, respectivly.

$$\begin{array}{rl}
\Sigma_{L_\alpha} = \{ & makeinstr : BIT^k \times INSTR \\
& makeprog : ADDR \times INSTR \to PROG \quad \} \\
\Gamma_{L_\alpha} = \{ & start : PROG \times ADDR \\
& next : ADDR \to ADDR \\
& addr_instr : \mathbb{N} \times PROG \to ADDR \\
& get_instr : ADDR \times PROG \to INSTR \quad \}
\end{array}$$

For the interpretation of the Control Flow see Figure 10, section 4.1.

A.2.2 The operational semantics

The operational semantics uses the sorts QUAD and DOUBLE which are 64-bit sequences. It is not necessary to introduce or distinguish these sorts, but it makes the specification more readable. Furthermore, we use the sorts BYTE and LONG, which are isomorphic to BIT^8 and BIT^{32}, respectively. Operations on QUADs and DOUBLEs are the same as those defined by BB.
The dynamic functions of L_α are:

$$\begin{array}{rl}
reg: & BIT^6 \to QUAD \cup DOUBLE \\
content: & ADDR \to BYTE \\
PC: & ADDR
\end{array}$$

Furthermore: $IP = get\_instr(PC, prog)$

The specification uses the following macro definitions Θ_{L_α}: (for definitions of *content* and *reg* see Figure 11)

$$\begin{array}{lcl}
\text{BYTE} & \widehat{=} & \text{BIT}^8 \\
\text{QUAD} & \widehat{=} & \text{BIT}^{64} \\
\text{DOUBLE} & \widehat{=} & \text{BIT}^{64}
\end{array}
\qquad
\begin{array}{lcl}
Reg_{quad}(X) & \widehat{=} & reg(0.X) \\
Reg_{flt}(X) & \widehat{=} & reg(1.X)
\end{array}$$

The *reg* macros reflect the two different kinds of registers. Furthermore, a series of macros is used to define the bit-sequences and to interpret them. These macros define the bit-sequences representing instructions in a symbolic way. The expansion of these terms always leads to bit-sequences. The way how these bit-sequences are obtained is described in the instruction manual. We demonstrate this by the ADD-instruction (See figure 18). For this article it is sufficient to know that there is an expansion according to the instruction manual.

$$\begin{array}{l}
LDA: \quad BIT^5 \times BIT^{16} \times BIT^5 \to INSTR \\
LDA(ireg1, disp, ireg2) \;\; Load\ address\ (ireg2 + disp)\ to\ ireg1
\end{array}$$

31					26	25				21	20				16	15			12	11						5	4				0
0	1	0	0	0	0	a	a	a	a	a	b	b	b	b	b	0	0	0	0	0	1	0	0	0	0	0	c	c	c	c	c
opcode						register a					register b					type + function code											register c				

Figure 18: Bit-sequence representing the ADD-instruction of type QUAD

$$
\begin{array}{l}
LDQ: \quad BIT^5 \times BIT^{16} \times BIT^5 \to INSTR \\
LDQ(reg1, disp, ireg2) \quad Load\ integer\ from\ memory\ (ireg2 + disp)\ to\ reg1 \\
LDT: \quad BIT^5 \times BIT^{16} \times BIT^5 \to INSTR \\
LDT(reg1, disp, ireg2) \quad Load\ float\ from\ memory\ (ireg2 + disp)\ to\ reg1 \\
STQ: \quad BIT^5 \times BIT^{16} \times BIT^5 \to INSTR \\
STQ(reg1, disp, ireg2) \quad Store\ integer\ reg1\ to\ memory\ (ireg2 + disp) \\
STF: \quad BIT^5 \times BIT^{16} \times BIT^5 \to INSTR \\
STF(reg1, disp, ireg2) \quad Store\ float - pointreg1\ to\ memory\ (ireg2 + disp) \\
ADD: \quad BIT^5 \times BIT^5 \times BIT^5 \times BIT \to INSTR \\
ADD(reg1, reg2, reg3, type) \quad Add\ reg1 + reg2\ into\ reg3,\ all\ of\ type\ 'type' \\
BR: \quad BIT^5 \times BIT^{21} \to INSTR \\
BR(ireg, offs) \quad Branch\ unconditionally\ to\ offset\ 'offs',\ relative\ to\ PC,\ ireg = PC \\
BEQ: \quad BIT^5 \times BIT^{21} \to INSTR \\
BEQ(ireg, offs, cond) \quad Check\ condition\ 'equal'\ on\ ireg,\ branch\ conditionally \\
BLT: \quad BIT^5 \times BIT^{21} \to INSTR \\
BLT(ireg, offs, cond) \quad Check\ condition\ 'less - than'\ on\ ireg,\ branch\ conditionally \\
\ldots \\
JMP: \quad BIT^5 \times BIT^5 \to INSTR \\
JMP(ireg1, ireg2) \quad Jump\ to\ address\ contained\ in\ ireg2,\ ireg1 = PC
\end{array}
$$

$$
\begin{array}{ll}
\text{IntTest (EQ, op)} & = (\ \text{op} =_I 0_2^{64}\ ) \\
\text{IntTest (LT, op)} & = (\ (op)\langle 63\rangle = 1_2\ ) \\
\text{IntTest (LE, op)} & = (\ \text{IntTest (LT, op)} \vee \text{IntTest (EQ, op)}\ ) \\
\ldots &
\end{array}
$$

Furthermore, the following macros are used in the transition rules:
$Sext_n : BIT^n \to QUAD$ sign extends its argument to a 64-bit integer:
$Sext_n(X) \mathrel{\widehat{=}} \underbrace{X_{n-1}.\ldots.X_{n-1}}_{64-n\ times}.X$

$LogShift_L : QUAD \times BIT^6 \to QUAD$ shifts logically the first argument by the amount of the second argumentto the left. It is defined by 2^k macros $(0 \leq k \leq 63)$.
$LogShift_L(X, k) \mathrel{\widehat{=}} X\langle 0 : 64 - k\rangle \dot{\ } \langle \underbrace{0, \ldots, 0}_{k} \rangle$

$AND_n : BIT^n \times BIT^n \to BIT^n$ defines the bitwise *and* for sequences of n-bits:
$AND_1(0, b) \mathrel{\widehat{=}} 0$
$AND_1(1, b) \mathrel{\widehat{=}} b$
$AND_n(0.X, b.Y) \mathrel{\widehat{=}} 0.AND_{n-1}(X, Y)$

$AND_n(1.X, b.Y) \mathrel{\widehat{=}} b.AND_{n-1}(X, Y)$

In arithmetic operations $(ADD, \ldots)$ the type of the operation is destinguished by one bit, denoted by: $Q \mathrel{\widehat{=}} 1, F \mathrel{\widehat{=}} 0$
Finally we give some transition rules. For simplicity we omit the exception handling:

STORE — ASM4

if $PC = \ STQ(ra, disp, rb)$ **then**
$\quad \overline{content}_8(Reg_{quad}(rb) \oplus_A Sext_{16}(disp)) := Reg_{quad}(ra)$
$\quad PC := next(PC)$
endif
endif
if $PC = \ STT(ra, disp, rb)$ **then**
$\quad \overline{content}_8(Reg_{quad}(rb) \oplus_A Sext_{16}(disp)) := Reg_{fltd}(ra)$
$\quad PC := next(PC)$
endif

LOAD — ASM5

if $PC = \ LDQ(ra, disp, rb)$**then**
$\quad Reg_{quad}(ra) := content(Reg_{quad}(rb) \oplus_A Sext_{16}(disp))$
$\quad PC := next(PC)$
endif
if $PC = \ LDT(ra, disp, rb)$**then**
$\quad Reg_{fltd}(ra) := content(Reg_{quad}(rb) \oplus_A Sext_{16}(disp))$
$\quad PC := next(PC)$
endif

if $PC = \ LDA(ra, disp, rb)$ **then**
$\quad Reg_{quad}(ra) := Reg_{quad}(rb) \oplus_A Sext_{16}(disp)$
$\quad PC := next(PC)$
endif

Arithmetic operations are defined analogously:

`ADD` — ASM6

if $PC = \ ADD(ra, rb, rc, type)$ **then**
$\quad$ **if** $type = F$ **then** $Reg_{fltd}(rc) := Reg_{fltd}(ra) \oplus_F Reg_{fltd}(rb)$
$\qquad PC := next(PC)$
$\quad$ **if** $type = Q$ **then** $Reg_{quad}(rc) := Reg_{quad}(ra) \oplus_Q Reg_{quad}(rb)$
$\qquad PC := next(PC)$
$\quad$ **endif**
endif

BRANCH ASM7

if $PC = BR(ra, disp)$ **then**
$Reg_{quad}(ra) := next(PC)$
$PC := PC \oplus_A 4 \oplus_A LogShift_L(Sext_{21}(disp), 2)$
endif

if $PC = BEQ(ra, disp)$ **then**
if $IntTest('EQ', Reg_{quad}(ra)$ **then** $PC := next(PC) \oplus_A LogShift_L(Sext_{21}(disp), 2)$
else $PC := next(PC)$
endif

JUMP ASM8

if $PC = JMP(ra, rb)$ **then**
$Reg_{quad}(ra) := next(PC)$
$PC := (Reg_{quad}(rb) And_Q 11_2^{62} 00$
endif

A.3 Dec-Alpha Basic Block Graphs BB_α

The operational semantics of DEC-Alpha basic block graphs can be defined uniquely from the operational semantics of the basic block graphs and the DEC-Alpha processor family. Consider DEC-Alpha basic block graph π and its abstract state machine $\mathcal{A}$. The sorts of $\mathcal{A}$ contain the sorts of the target machine (except the instruction set which is partially different). Additionally, it contains the same sort *LABEL* as the basic block graphs. The signature of $\mathcal{A}$ contains the same dynamic functions as the abstract state machines for the DEC-Alpha except the instruction pointer. Instead, it contains the instruction pointer, the basic block pointer and the procedure pointer of the basic block graph. *loc* and *glob* are defined by the macros $loc \hat{=} Reg_{quad}(R1)$ and $glob \hat{=} Reg_{quad}(R2)$, respectively.
The transition rules are the same as on the DEC-Alpha except the *jump* instruction.
BB_{Alpha} cointains amoung other the following transition rules:

if $IP = jump(B(ra, disp), L)$ **then**
$BP := L$
$PC := 0$
endif

if $IP = jump(JMP(ra, rb), L)$ **then**
$BP := L$
$PC := 0$
endif

if $IP = jump(BR(ra, disp, cond), L)$ **then**
if $IntTest(Reg_{quad}(ra), cond)$ **then**
$BP := L$
$PC := 0$
else
$PC := next(PC)$
endif
endif

Remark: Other processors than DEC-Alpha may contain a status register and conditional jumps are based on whether some particular bits are set or not. Then, b is omitted, *type* are comparisons of checking whether a certain status bit is set or not, and the test *content*(b) *type* is just replaced by *type*. ■

B Error detection

Detecting faulty parts in code generator specifications usually is a tedious job. Sometimes testing the generated code with sample input does a good job, but especially the transformation to concrete machine instructions is highly erroneous, because the effects of machine dependent data manipulations and side effects are often not easy to realize.
The following example is taken from our own development cycle of a complete compiling specification. It shows the faulty variant of our example of section 6, that occurred while developing with our students:

$$intconst_{i32} \longrightarrow X \left\{ \begin{array}{l} LDA\ (T1, i32\langle 0:15\rangle, R31) \\ LDAH\ (X, i32\langle 16:31\rangle, T1) \end{array} \right\} \tag{22}$$

Intuitively our sequence seems to do a good job: The low-word is first loaded into *T1* (instruction 1), and then the high-word is shifted by the length of a word and added to the low-word in *T1*. We try now to prove the local correctness of rule 22, i.e. we try to prove the

Presumption 28 (Local Correctness of Rule 22) Let $\pi, \pi \in L$ be arbitrary programs with $\pi \rhd \pi'$, $\mathcal{A}_\pi$ and $\mathcal{A}_{\pi'}$ their ASMs in $\mathbb{A}_L$, $q \in Q_\pi$ a state with $[\![IP]\!]_q = instr$ where rule 22 is applied onto *instr* to obtain π from π', σ be the corresponding BE-substitution, $q' \in Q_{\pi'}$ be a state such $[\![IP]\!]_{q'} = LDA(\sigma(T1), i32.\mathsf{L}, R31, \mathsf{L})$, and $q'' \in Q_{\pi'}$ be the state such that $q' \rightarrow_{\pi'} q_1 \rightarrow_{\pi'} \rightarrow_{\pi'} q''$. Then, for any $\rho \subset Q_\pi \times Q_{\pi'}$ satisfying the requirements defined in subsection 3.3 $(q, q') \in \rho$ implies $[\![eval(intconst_{i32})]\!]_q = [\![Reg_{quad}]\!]_{q''}(\sigma(X))$.

For proving the correctness we proceed as in section 6. We use the same notations as in the proof of lemma 26. Analogous to this proof, we show (15) and (16). By the rule for *LDA*, the update $Reg_{quad}(\sigma(X)) := Reg_{quad}(\sigma(T1)) \oplus_I LogShift_L(Sext_{16}(i32.\mathsf{H}), 16)$ is performed on q_1. With (16) and (14) we obtain:

$$[\![Reg_{quad}]\!]_{q''}(\sigma(X)) = s_l^{48} \circ (i32)\langle 0:15\rangle \oplus_I s_h^{32} \circ (i32)\langle 16:31\rangle \circ 0^{16}.$$

However, $i32 = s_l^{48} \circ (i32)\langle 0:15\rangle \oplus_I s_h^{32} \circ (i32)\langle 16:31\rangle \circ 0^{16}$ only if $s_l = 0$. Therefore, the above rule is faulty if 16-th bit of $i32$ is set. This error is very to find with software testing, because it only occurs, if bit 15 of the desired integer constant is non-zero. A solution is to compensate the sign extension with an arithmetic operation or to zero the sign extended bits after the first instruction. This version was chosen for rule 6 in subsection 4.3.

C Notations

Signatures are denoted by capital greek letters. Sorts and universes are denoted by capital letters; usually taken from the end of the alphabet. Mappings and homomorphisms and denoted by lower case greek letters. Algebras and ASMs are denoted by calligraphic letters. Symbols defined by signatures are denoted by lower case letters. An additional notation is that of indexing ASM. If an ASM $\mathcal{A}_i$ is indexed with index i, then Σ_i refers to the signature, Q_i to the states, S_i to the sorts, $\rightarrow_i$ to the transition relation, and I_i to the initial states of $\mathcal{A}_i$, respectively. F_i refers to the set of final states of $\mathcal{A}_i$.Table 7–Table 10 summarize notations commonly used in this article.

$t[o]$	subterm of t at occurence o	Subsection 2.1
$t[o/t']$	the term obtained from t by replacing $t[o]$ by t'	Subsection 2.1
$T(\Sigma)$	set of terms over signature Σ	Subsection 2.1
$T(\Sigma, V)$	set of terms over signature Σ and variables V	Subsection 2.1
$[\![t]\!]_{\mathcal{A}}$	interpretation of term t in algebra $\mathcal{A}$	Subsection 2.1
$\mathcal{T}(\Sigma)$	term algebra of terms over signature Σ	Subsection 2.1
$\sigma = [x_1/t_1] \cdots [x_n/t_n]$	substitution of variables by terms	Subsection 2.1
$t[u/u']$	the term t where sub-term u is replaced by u'	Subsection 2.1
$\mathcal{A}\|_{\Sigma}$	restriction of an algebra	Subsection 2.1
$t_1 \hat{=} t_2$	term-rewrite rule	Subsection 2.1
$\hookleftarrow$	rewrite relation defined by a TRS	Subsection 2.1
$NF_R(t)$	normal form of term t w.r.t. a noetherian and confluent TRS	Subsection 2.1
$i, i', i_1, \ldots$	initial states of an ASM	Subsection 2.2
$q, q', q_1, \ldots$	states of an ASM	Subsection 2.2
$\Xi, \Xi', \Xi_1, \ldots$	static functions of an ASM	Subsections 2.2, 3.2
$\mathcal{X}, \mathcal{X}', \mathcal{X}_1, \ldots$	static algebra of an ASM	Subsections 2.2, 3.2
$\xi, \xi', \xi_1, \ldots$	ASM-homomorphisms	Subsection 2.2
$U \sqsubseteq V$	$[\![U]\!]_{\mathcal{T}(\Sigma)} \subseteq [\![V]\!]_{\mathcal{T}(\Sigma)}$	Subsection 2.1

Table 7: Notations defined in Section 2

Σ_L	program structure of language L	Subsection 3.1
Γ_L	control structure of language L	Subsection 3.1
S_L	sorts of language L	Subsection 3.1
$INSTR$	sort of instructions	Subsection 3.1
$PROG$	sort of programs	Subsection 3.1
$\mathcal{I}_L$	interpretation of control and program structure of language L	Subsection 3.1
Υ_L	signature of instructions of language L	Subsection 3.1
$\pi, \pi, \ldots$	programs	Subsection 3.1
IP	instruction pointer	Subsection 3.2
$Stat_L$	static part of an operational semantics of L	Subsection 3.2
$\Psi, \Psi', \Psi_1, \ldots$	static functions of an operational semantics of L not used by the control and program structure	Subsection 3.2
Dyn_L	dynamic part of an operational semantics of L	Subsection 3.2

Table 8: Notations defined in Section 3 (1)

$\Delta, \Delta', \Delta_1, \ldots$	dynamic functions of an operational semantics of L	Subsection 3.2
$\Omega, \Omega', \Omega_1, \ldots$	observable dynamic functions of an operational semantics of L	Subsection 3.2
$\Theta, \Theta', \Theta_1, \ldots$	signature of macros of an operational semantics of L	Subsection 3.2
$\mathbb{A}_L$	operational semantics of a language L	Subsection 3.2
$\mathcal{A}_\pi$	ASM of π in $\mathbb{A}_L$	Subsection 3.2
$qq, qq', \ldots$	computation sequence of a program $\pi \in L$	Subsection 3.2
B_π	behavior of program π	Subsection 3.2
$\sim_\Omega$	Ω-equivalence relation	Subsection 3.2
$[q]_\Omega$	Ω-equivalence class of state q	Subsection 3.2
ob_{qq}	observable behavior of qq	Subsection 3.2
OB_π	observable behavior of program π	Subsection 3.2
$jj, ll, jj', ll', \ldots$	witnesses of observable behavior of qq	Subsection 3.2
$\zeta, \zeta', \zeta_1, \ldots$	L-semantics monomorphisms	Subsection 3.2
$\rho, \hat{\rho}, \rho', \ldots$	relations between Ω-equivalence classes or states	Subsection 3.2
$\mathcal{C}$	compiling relation	Subsection 3.3

Table 9: Notations defined in Section 3 (2)

$JUMP$	sort of jump instructions in basic block graphs	Definition 14
$BLOCK$	sort of basic blocks	Definition 14
$EXPR$	sort of expressions	Definition 14
$LABEL$	sort of labels	Definition 14
$ADDRESS$	sort of addresses	Definition 14, 15
$VALUE$	sort of values	Definition 14, 15
BP	block pointer	Definition 14
PC	program counter	Definition 14, 15
IM	signature of instruction macros	Subsection 4.3
$t \to X; \{m_1, \ldots, m_n\}$	back-end term-rewrite rule	Definition 18
$\sigma_{\pi, instr, o}$	register assignment for application of a rule on *instr* at occurence *o*	Definition 18
$\triangleright$	rewrite relation of term-rewrite systems for back-ends	Definition 18
rule	rule annotation	Definition 19
regassign	register assignment	Definition 19

Table 10: Notations defined in Section 4

References

[Blum and Kannan 1995] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[Börger and Rosenzweig 1992] E. Börger and D. Rosenzweig. The WAM-definition and Compiler Correctness. Technical Report TR-14/92, Dip. di informatica, Univ. Pisa, Italy, 1992.

[Börger et al. 1994] Egon Börger, Igor Durdanovic, and Dean Rosenzweig. Occam: Specification and Compiler Correctness.Part I: The Primary Model. In U. Montanari and E.-R. Olderog, editors, *Proc. Procomet'94 (IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi)*. North-Holland, 1994.

[Börger and Durdanovic 1996] E. Brger and I. Durdanovic. Correctness of compiling occam to transputer. *The Computer Journal*, 39(1):52–92, 1996.

[Brown et al. 1992] D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *Compiler Compilers 92*, volume 641 of *Lecture Notes in Computer Science*, 1992.

[Buth et al. 1992] B. Buth, K.-H. Buth, M. Fränzle, B. v. Karger, Y. Lakhneche, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[Buth and Müller-Olm 1993] Bettina Buth and Markus Müller-Olm. Provably Correct Compiler Implementation. In *Tutorial Material – Formal Methods Europe '93*, pages 451–465, Denmark, April 1993. IFAD Odense Teknikum.

[Diehl 1996] S. Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, Universität Saarbrücken, 1996.

[Dold and Gaul 1996] A. Dold and T.S. Gaul. Local correctness of term rewrite based code generators. Working paper, University of Karlsruhe/Ulm, September '96, 1996.

[Emmelmann 1992] H. Emmelmann. Code selection by regularly controled term rewriting. In R. Giegerich and S.L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, Workshops in Computing. Springer-Verlag, 1992.

[Espinosa 1995] David A. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[Gaul and Zimmermann 1995] T.S. Gaul and W. Zimmermann. An Evolving Algebra for the Alpha Processor Family. Verifix Working Paper [Verifix/UKA/4], University of Karlsruhe, 1995.

[Gurevich 1995] Y. Gurevich. Evolving Algebras: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Gurevich and Huggins 1993] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In *CSL '92*, volume 702 of *LNCS*, pages 274–308. Springer-Verlag, 1993.

[Hoare et al. 1993] C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.

[McCarthy and Painter 1967] J. McCarthy and J.A. Painter. Correctness of a compiler for arithmetical expressions. In J.T. Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

[Moore 1989] J. Strother Moore. System verification. *Journal of Automated Reasoning*, 5(4):409–410, December 1989.

[Mosses 1982] P. D. Mosses. Abstract semantic algebras. In D. Bjørner, editor, *Formal description of programming concepts II*, pages 63–88. IFIP IC-2 Working Conference, North Holland, 1982.

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

[Müller-Olm 1995] Markus Müller-Olm. An Exercise in Compiler Verification. Internal report, CS Department, University of Kiel, 1995.

[Müller-Olm 1996] Markus Müller-Olm. *Modular Compiler Verification.* PhD thesis, Techn. Fakultät der Christian-Albrechts-Universität, Kiel, June 1996. Erscheint als LNCS Band im Springer-Verlag.

[Nymeyer and Katoen 1996] Albert Nymeyer and Joost-Pieter Katoen. Code Generation based on formal BURS theory and heuristic search. Technical report inf 95-42, University of Twente, 1996.

[Nymeyer et al. 1996] Albert Nymeyer, Joost-Pieter Katoen, Ymte Westra, and Henk Alblas. Code Generation = A* + BURS. In Tibor Gyimothy, editor, *Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science*, pages 160–176. Springer-Verlag, April 1996.

[Palsberg 1992] J. Palsberg. An automatically generated and provably correct compiler for a subset of ada. In *IEEE International Conference on Computer Languages*, 1992.

[Paulson 1981] L. Paulson. *A compiler generator for semantic grammars.* PhD thesis, Stanford University, 1981.

[Pierantonio and Kutter 1997] A. Pierantonio and P. W. Kutter. Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, this volume, 1997.

[Proebsting 1995] Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

[Sites 1992] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992.

[Tofte 1990] M. Tofte. *Compiler Generators.* Springer Verlag, 1990.

[Waite and Goos 1984] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer Verlag, 1984.

[Wallace 1995] C. Wallace. The Semantics of the C++–Programming Language. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[Wand 1984] M. Wand. A semantic prototyping system. *SIGPLAN Notices*, 19(6):213–221, June 1984. SIGPLAN 84 Symp. On Compiler Construction.