On the Correctness of Transformations in Compiler Back-Ends

Wolf Zimmermann

Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg
06099 Halle/Saale, Germany
zimmer@informatik.uni-halle.de

Abstract. This paper summarizes the results on the correctness of the transformations in compiler back-ends achieved in the DFG-project *Verifix*. Compiler back-ends transform intermediate languages into code of the target machine. Back-end generators allow to generate compiler back-ends from a set of transformation rules. This paper focuses on the correctness of these transformation rules and on the correctness of the whole transformation stemming from the transformation rules.

1 Introduction

Verification of software systems is often done on the level of the source language. However, it is the binary code generated by a compiler that is really executed. Thus, unless the compiler guarantees the correctness of its compilation, a verification of the source code doesn't say anything about the correctness of the binary code. In order to avoid bugs at the binary level, either the binary code has to be verified directly or source code has to be verified and compiled into binary code by a correct compiler. Compiler bugs are more frequent than expected (see e.g. the Borland Pascal compiler Bug List and the Java Bug Database).

The aim of the DFG-project[1] *Verifix* was to develop approaches for the construction of *verifying* compilers. These guarantee that if a target program τ is generated from a source program σ, then τ is a correct translation of σ. The approaches should work for *realistic* source languages, as e.g. defined by ISO-standards, and *realistic* target languages as defined e.g. by assembly languages of industrial processors. Apart from these goals, a verifying compiler should generate code that is as good in the sense of time and space as it could be generated by a non-verifying compiler. *Verifix* achieved these goals by using the classical compiler architecture which is well-established since more than 25 years. In particular, this decision implies that there are no restrictions on the choice of source and target languages. Although not considered as a part of the project, verification shouldn't restrict the use of optimizing transformations.

This paper focuses on the correctness of the transformations in the code generation phase, i.e. how to verify the transformation from intermediate code to binary code. [24,21] show how this part is embedded into a whole verifying

[1] *Verifix* was supported by DFG grants Go 323/3-3, He 2411/2-3, La 426/15-3.

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 74–95, 2006.

compiler. We consider classical basic-block oriented intermediate languages and sequential register based target machines. This also includes pipelined and superscalar processors where scheduling is done by hardware. However, we do not consider optimizations on this level. The requirements are the same as used by back-end generators such as BEG [14,13]. They generate compiler back-ends from transformation rules specified as a special class of term-rewrite rules. Code-generation consists of two phases: code selection and assembly. Code selection replaces all instructions except jumps in the basic blocks by machine instructions (in their binary format). Assembly linearizes the basic block graphs and thereby generates jump instructions (again in binary format). Tools such as BEG specify code selection. We finally assume that the formal operational semantics of the intermediate language and the target language is given as Abstract State Machines. The contribution of *Verifix* is that from a practical viewpoint there are just two cases to consider (transformation of side-effect free expressions and instructions with side-effects) and for each of these cases there is a simple mechanizable proof strategy to prove the correctness of the corresponding transformation rule.

The paper is organized as follows: Section 2 introduces Abstract State Machines as required for this paper. Section 3 sketches code selection by term-rewrite systems. Section 4 introduces the notion of correctness and applies it to the correctness of transformation specified by term-rewrite systems. Section 5 discusses the two proof strategies and argues why these two are sufficient. Section 6 introduces the task of assembly, the transformations, and the approach for correctness proofs. Section 7 discusses related work and Section 8 presents our conclusions.

2 Abstract State Machines and Language Semantics

We first define the notion of Abstract State Machines. For more details, we refer to the Lipari-Guide [25] and the ASM 1997 Guide [26]. An *Abstract State Machine* (ASM) is a tuple $(\Sigma, \Phi_{\text{Init}}, \mathit{Trans})$ where Σ is a signature, Φ_{Init} is a set of Σ-formulas (the *initial conditions*), and *Trans* is a finite set of *transition rules*. The set of *states* is the set $\text{Alg}(\Sigma)$ of Σ-algebras. Σ-terms are defined as usual. A state q is *initial* iff q is a model of Φ_{Init} in the sense of logic, denoted as $q \models \Phi_{\text{Init}}$. In this article, we use order-sorted partial Σ-algebras.

Notation: Sorts are denoted by capital letters, function symbols always start with a lower-case letter. $S_1 < S_2$ denotes that S_1 is a sub-sort of S_2. The symbol $f : T_1 \times \ldots \times T_n \to T$ denotes a function symbol representing a total function and $f : T_1 \times \ldots \times T_n \to ?T$ denotes a function symbol representing a partial function. $[\![\cdot]\!]_q$ denotes the interpretation function of symbols of Σ in Σ-algebra q. This notion is extended as usual to Σ-terms. $t \in S$ is satisfied iff t is a term of sort S. $t[x/t']$ denotes the term t where each occurrence of variable x is substituted by term t'. This can be extended as usual to Σ-formulas (here, only free variables are substituted) and to transitions.

Transition rules are specified as in [26]. In this paper, we use the following kinds of transition rules: **if** φ **then** *Transitions* **endif**
if φ **then** $\mathit{Transitions}_1$ **else** $\mathit{Transitions}_2$ **endif**

Table 1. Typical Instructions of an Intermediate Language

$:=_I$:	$EXPR \times EXPR \to ASSIGN$		integer assignment
$:=_A$:	$EXPR \times EXPR \to ASSIGN$		address assignment
$goto$:	$LABEL \to JUMP$		unconditional jump
$+_I$:	$EXPR \times EXPR \to EXPR$		integer addition
$+_A$:	$EXPR \times EXPR \to EXPR$		address addition
ld_I:	$EXPR \to EXPR$		read integer from memory
ld_A:	$EXPR \to EXPR$		read address from memory
int_{ck}:	$\to EXPR$		k-bit integer constant
$addr_{ck}$:	$\to EXPR$		k-bit address constant

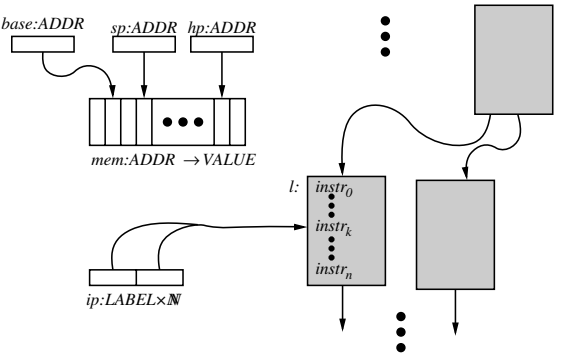

Fig. 1. State Space of an Example Intermediate Language

where φ is formula of many-sorted first order predicate logic and *Transitions* is a set of transitions of one of the above forms or an update $f(t_1, \ldots, t_n) := t$. Such an update changes the interpretation of f: if the state transition from state q to state q' executes an update $f(t_1, \ldots, t_n) := t$ then

$$[\![f]\!]_{q'}(a_1, \ldots, a_n) = \begin{cases} [\![t]\!]_q & \text{if } a_1 = [\![t_1]\!]_q, \ldots, a_n = [\![t_n]\!]_q \\ [\![f]\!]_q(a_1, \ldots, a_n) & \text{otherwise} \end{cases}$$

The first kind of transition rule executes the transition rules of *Transitions* in a state q iff $q \models \varphi$. Similarly, the second kind of transition rule executes the transition rules of $Transitions_1$ in a state q iff $q \models \varphi$. Otherwise, the transition rules of $Transitions_2$ are executed in a state q. Macros can be used to abbreviate updates and terms. It is just a parameterized textual substitution mechanism. A macro definition has the form $macro(par_1, \ldots, par_k) \triangleq text$. If in an ASM-specification a term $macro(arg_1, \ldots, arg_k)$ is used, then it is textually replaced by $text[par_1/arg_1, \ldots, par_k/arg_k]$. A function is *dynamic* iff its interpretation may change, otherwise a function is called *static*. Note that a function may be dynamic because of explicit updates in transition rules or its definition depends on dynamic functions.

Abstract State Machines can be viewed as state transition systems. A *state transition system* is a triple $(Q, I, \to)$ where Q is the (possibly infinite) set of states, $I \subseteq Q$ is the set of initial states and $\to \subseteq Q \times Q$ is the state transition

relation. For an ASM $(\Sigma, \Phi_{init}, \mathit{Trans})$, it is $Q = \mathrm{Alg}(\Sigma)$ the set of Σ-algebras, $I = \{q \in Q : q \models \Phi_{init}\}$ the set of Σ-algebras satisfying the initial conditions, and *Trans* describes the state transition relation as above.

Example 1. The main principles for designing a dynamic semantics of programming languages are: First, define the (abstract) syntax and the state space and then define the transition rules for each class of instructions. Here, we only give a fragment for an intermediate language based on basic blocks. A *program* π is a set of procedures, a *procedure* p is a basic block graph, i.e., a labeled directed graph (BB, E, lab) with a designated initial basic block i where $lab : BB \to LABEL$, $LABEL$ is the sort of symbolic labels, each *basic block* $bb \in BB$ is a finite sequence of instructions containing no *jump*-instruction except possibly the last instruction, and there is an edge $(bb_1, bb_2) \in E$ iff the jump instruction of bb_1 has the jump target $lab(bb_2)$. For simplicity, we assume that each basic block in a program has a unique label. Note that this implies that the inverse function $lab^{-1} : LABEL \to BB$ is well-defined. Table 1 shows the signature of some typical instructions. Each instruction is a term over this signature and can therefore be viewed as a tree.

The state space (see Fig. 1) consists of the memory *mem* of the target machine, $ADDR$ and $VALUE$ are the sort of addresses and the sort of values that can be stored in the memory, a stack pointer *sp*, a base address *base*, a heap pointer *hp*, and an instruction pointer *ip*. $ip = (l, k)$ means that the instruction to be executed next is the k-th instruction of the basic block *bb* with label l. Note that memory mapping is often part of the intermediate code generation. In particular the execution environment is already mapped into memory.

Fig. 2 shows the definitions of expression evaluation, the ASM state transition rules for integer assignment, and the state transition rules for jumps. Note that *eval* is a dynamic function. The function $SExt_k(x)$ is a signed extension of the k-bit sequence x to a 64-bit integer or a 64-bit relative address[2]. The function *instr* computes the instruction corresponding to an instruction pointer. The macro *ip* **is** T defines the class of instruction, the macro *Proceed* advances the instruction pointer if it has not yet reached the end of a basic block (k-tuples are denoted as $(t_1, \ldots, t_k)$ and the projection to the i-th element of a tuple t is denoted by $t \downarrow_i$), the functions $lhs, rhs : LABEL \times \mathbb{N} \to ?EXPR$ compute the left and right hand sides of an assignment, respectively, and the function $target : LABEL \times \mathbb{N} \to ?LABEL$ computes the jump target.

The next example contains some state transition rules from the DEC-Alpha machine language specification, for a complete specification see [16].

Example 2. All necessary information for the formal semantics of machine languages is provided by the processor manual. Usually, the dynamic semantics is given by register transfers which already are very close to updates of ASMs. The semantics can immediately be provided for the binary format of instructions

[2] Here, we use the DEC-Alpha as target machine which has 64-bit integer and address arithmetic. If a 32-bit machine is used then the definition of the intermediate language has to be changed accordingly.

$eval : EXPR \to VALUE$
$eval(int_{ck}) = SExt_k(c)$
$eval(addr_{ck}) = base \oplus_A SExt_k(c)$
$eval(ld_I(x)) = mem(eval(x))$
$eval(ld_A(x)) = mem(eval(x))$
$eval(x +_I y) = eval(x) \oplus_I eval(y)$
$eval(x +_A y) = eval(x) \oplus_A eval(y)$
$\oplus_I, \oplus_A$ is integer/address addition on the target machine
$instr((l,k)) = lab^{-1}(l)_k$

if ip **is** $ASSIGN$ **then**
  $mem(eval(lhs(ip))) := eval(rhs(ip))$
  $Proceed$
endif
if ip **is** JMP **then**
  $ip := (target(ip), 0)$
endif

Macros:
ip **is** $T \triangleq instr(ip) \in T$
$Proceed \triangleq ip := (ip\downarrow_1, ip\downarrow_2 +1)$

Fig. 2. Expression Evaluation and State Transitions

using adequate access functions, cf. Fig. 3. They use sorts of bit sequences of a certain length: $QUAD$, $LONG$, $WORD$, $TFCODE$, $BYTE$, $OPCODE$, and $RADDR$ denote the sorts of 64-bit sequences (quad integers and addresses), 32-bit sequences (long integers), 16-bit sequences (words), 11-bit sequences (type and function code[3]), 8-bit sequences (bytes), 6-bit sequences (operation codes), and 5-bit sequences (register addresses), respectively. Bit sequences may denoted in hexadecimal, decimal, binary notation (as in C), or explicitly as lists (denoted by $[x_1, \ldots, x_n]$). The state space of the DEC-Alpha is also shown in Fig. 3. The function reg represents the 32 integer registers. The 32 floating point registers $freg$ are just added for completeness but are not important for this paper. Note that the memory mem_α is byte-oriented and addressed by quads. The macros for mem define how to read and write quads into the byte oriented memory mem_α. Here, l_i denotes the i-th element of a list l (l_0 is the first element), $split_n(l)$ splits a list l into a list of lists of length n (the last list may contain less than n elements), and $concat(l)$ concatenates all lists in a list l of lists. Note that l_i is defined iff i is less than the length of l. The macros for reading and writing the byte-oriented memory specify therefore how to map the memory mem of the intermediate language to the memory mem_α of the target languages. It only requires an identification of sorts and operations of the static part of the intermediate language specification with those used by the machine language specification. E.g. the sorts INT and $ADDR$ of the intermediate language are both identified with $QUAD$. The operations $\oplus_I$ and $\oplus_A$ used in the specification for the intermediate language are for the purpose of the paper additions on integers and addresses. They are both mapped to $\oplus_Q$-operation on quads. Furthermore, the program is in the memory. Therefore, the program counter pc contains the address of the next instruction to be executed.

Fig. 4 shows some state transitions of the DEC-Alpha Machine. The macros rb and rc are defined analogously as ra using functions $regb$ and $regc$ accessing the other registers encoded in an arithmetic instruction. The second instruction shows an addition where the second operand is a 16-bit constant directly encoded in the instruction. LDQ loads a register ra using address register rb and relative address $disp$. LDA works similarly but does not access the memory and loads the content of rb plus the constant $disp$. The constant may be shifted to the higher

[3] In particular, this might also indicate whether the second operand is a constant directly encoded in the instruction.

State Space

$reg : RADDR \to QUAD$
$freg : RADDR \to QUAD$
$mem_\alpha : QUAD \to BYTE$
$pc : QUAD$

Auxiliary Functions:

$opcode : LONG \to OPCODE$ opcode
$rega : LONG \to ?RADDR$ register a
$type : LONG \to ?TFCODE$ operation type
$immed : LONG \to ?WORD$ constant operand
$disp : LONG \to ?WORD$ const. rel. address
$imbyte : LONG \to ?BYTE$ byte in ZAP-instr.

Macros

$mem(a) \triangleq concat([mem_\alpha(a), \ldots, mem_\alpha(a \oplus_Q 7)])$
$instr_\alpha(a) \triangleq concat([mem_\alpha(a), \ldots, mem_\alpha(a \oplus_Q 4)])$
$mem(a) := q \triangleq mem_\alpha(a) := split_8(q)_0$
$\vdots$
$mem_\alpha(a \oplus_Q 7) := split_8(q)_7$
pc **is** ADD $\triangleq$ $opcode(instr(pc)) = 010000$
pc **is** ADDQ $\triangleq$ pc **is** ADD$\wedge$ $type(instr(pc)) = 0x020$
$Proceed \triangleq pc := pc \oplus_Q 4$
$ra \triangleq rega(instr(pc))$

Fig. 3. State Space of the DEC-Alpha Processor Family and Some Macros

```
if pc is ADDQ then
   reg(rc) := reg(ra) ⊕Q reg(rb)
   Proceed
endif
if pc is ADDI then
   reg(rc) := reg(ra) ⊕Q SExt16(immed(pc))
   Proceed
endif
if pc is LDQ then
   reg(ra) := mem(reg(rb) ⊕ SExt16(disp))
   Proceed
endif
if pc is LDA then
   reg(ra) := reg(rb) ⊕ SExt16(disp)
   Proceed
endif
if pc is LDAH then
   reg(ra) := reg(rb) ⊕ LogShift(SExt16(disp), 16))
   Proceed
endif
if pc is STQ then
   mem(reg(rb) ⊕ SExt16(disp)) := reg(ra)
   Proceed
endif
if pc is ZAP then
   reg(rc) := zerobytes(reg(ra), imbyte(pc))
   Proceed
endif
if pc is SLL then
   reg(rc) := LogShift(reg(ra), immed(pc)⟨58 : 63⟩)
   Proceed
endif
if pc is BR then
   reg(ra) := pc ⊕ 4
   pc := pc ⊕Q 4 ⊕Q LogShift(SExt21(disp, 2))
endif
if pc is JMP then
   reg(ra) := pc ⊕ 4
   pc := reg(rb) ∧Q #fffffffc
endif
```

Fig. 4. Some State Transition Rules for the DEC-Alpha

16 bits. STQ is the instruction dual to LDQ and stores ra. The ZAP instruction explicitly sets some bytes in a quad word to zero. This is modeled by the function $zerobytes : QUAD \times BYTE \to QUAD$. The i-th byte in $zerobyte(q, b)$ is the zero byte iff the i-bit of b is 1, otherwise it is equal to i-th byte of q. The need of this instruction will become clear later (cf. Section 5). The SLL-instruction is the logical left shift. The last 6 bits of the register rb determine the number of bits to be shifted[4]. The jump instruction BR is a relative jump. Its operand is a 21-bit relative jump address directly encoded in the instruction. Since the alignment of instructions, it is multiplied by 4. The address of the instruction after the jump instruction is stored in register ra. The JMP instruction also stores in register ra this address. However, the jump target is contained as absolute address in register rb. Due to alignment restrictions, the last two bits are set explicitly to 0 using the bitwise conjunction $\wedge_Q$ for quadwords.

Note that these formulations can be extended in straightforward way to the semantics processor pipelines and instruction-level parallelism.

[4] $l\langle n : m\rangle$ denotes the sublist $[l_n, \ldots, l_m]$ of list l.

$$
\begin{array}{rcll}
X :=_I Y & \to \bullet & \{\mathsf{STQ}\ Y, (0)X\} & \text{Rule 1} \\
X :=_A Y & \to \bullet & \{\mathsf{STQ}\ Y, (0)X\} & \text{Rule 2} \\
addr_{c16} :=_I Y & \to \bullet & \{\mathsf{STQ}\ Y, (c16)R30\} & \text{Rule 3} \\
addr_{c16} :=_A Y & \to \bullet & \{\mathsf{STQ}\ Y, (c16)R30\} & \text{Rule 4} \\
int_{c16} & \to X & \{\mathsf{LDA}\ X, (c16)R31\} & \text{Rule 5} \\
int_{c32} & \to X & \{\mathsf{LDA}\ X, (c32.L)R31;\ \mathsf{ZAP}\ X, \#\mathrm{fc}, X;\ \mathsf{LDAH}\ X, (c32.H)X\} & \text{Rule 6} \\
addr_{c16} & \to X & \{\mathsf{LDA}\ X, (c16)R30\} & \text{Rule 7} \\
addr_{c32} & \to X & \{\mathsf{LDA}\ X, (c32.L)R31;\ \mathsf{ZAP}\ X, \#\mathrm{fc}, X;\ \mathsf{LDAH}\ X, (c32.H)X; \mathsf{ADDQ}\ X, R30, X\} & \text{Rule 8} \\
ld_I(Y) & \to X & \{\mathsf{LDQ}\ X, (0)Y\} & \text{Rule 9} \\
ld_A(Y) & \to X & \{\mathsf{LDQ}\ X, (0)Y\} & \text{Rule 10} \\
ld_I(c16) & \to X & \{\mathsf{LDQ}\ X, (c16)R31\} & \text{Rule 11} \\
ld_A(c16) & \to X & \{\mathsf{LDQ}\ X, (c16)R31\} & \text{Rule 12} \\
X +_I Y & \to Z & \{\mathsf{ADDQ}\ X, Y, Z\} & \text{Rule 13} \\
X +_A Y & \to Z & \{\mathsf{ADDQ}\ X, Y, Z\} & \text{Rule 14} \\
X +_I int_{c16} & \to Z & \{\mathsf{ADDI}\ X, \#c16, Z\} & \text{Rule 15} \\
X +_A addr_{c16} & \to Z & \{\mathsf{ADDI}\ X, \#c16, Z\} & \text{Rule 16}
\end{array}
$$

Fig. 5. Some Transformation Rules for Code Selection as Term Rewrite Rules

3 Generation of Compiler-Back Ends

Generating binary code from intermediate languages works in two phases: First, the *code selection* replaces the instruction sequence of each basic block by a machine language instruction sequence (except jumps). Second, the *assembly phase* linearizes the basic blocks and replaces jump instructions, if needed at all. We focus here on the code selection. The assembly phase is discussed in Section 6. The basic idea of term-rewriting in code selection is to apply term-rewrite rules of the form $t \to X$ or $t \to \bullet$ and associate with each rule a target code sequence $m_1, \ldots, m_n$ to be produced if the rule is applied. The first form of the rule is applied to expressions t while the latter is applied to instructions t. The term t may contain variables which are denoted by capital letters $X, Y, \ldots$ During application of a rule, each of these variables is associated with a register containing the value of the expression that is substituted for it. One may associate costs to each rule. A dynamic programming algorithm then determines the cost-optimal rule cover. However, this is not important for the purpose of correctness of the transformation.

Example 3. Fig. 5 shows some of the transformation rules for the code selection phase from our intermediate language programs to DEC-Alpha machine code. We used symbolic machine code for denoting the DEC-Alpha machine instructions. However, if register assignment is performed during code generation, this is just an abbreviation for the binary instruction format. $c_{32}.L$ and $c_{32}.H$ denote the lower and higher 16 bits of $c32$, respectively. It should be noted that some rules are specializations of other rules, e.g. rule 3 specializes rule 1. Usually, this is reasonable if one operand is a small constant (e.g. 16-bit constant). Special attention is required to understand rules 6 and 8. One could easily accidently forget the ZAP instruction. However, the LDA instruction automatically sign-extends the loaded constant (cf. Fig. 4), i.e. if bit 15 of $c32$ is 1, then all leading bits are set to 1. Hence, the following LDAH-instruction does not work properly because it expects that bits 16-63 are all 0. This is ensured by the ZAP-instruction. Observe that such kinds of compiler bugs are hard to identify. We found such a bug in a back-end written by one of our students using the proof strategies discussed in Section 5.

$addr_{28} :=_I ld_I(addr_{28}) +_I int_1$	(11)	$X = R2$	LDQ $R2, (28)R30$
$addr_{28} :=_I R2 +_I int_1$	(15)	$X = R2$ $Z = R2$	ADDI $R2, \#1, R2$
$addr_{28} :=_I R2$	(3)	$X = R2$	STQ $R2, (28)R30$
•			

Fig. 6. Code Generation for $addr_{0x001c} :=_I ld_I(addr_{0x001c}) +_I int_{0x0001}$

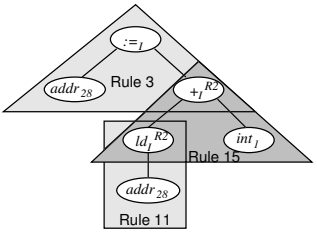

Fig. 7. Rule Cover corresponding to the Term-Rewrite of Fig 6

We show now how to apply the above rules to generate code. For each rule application a register has to be assigned to the RHS of a term-rewrite rule if it is a variable. Some of the registers are forbidden. In the DEC-Alpha processor family register $R31$ always contains 0 and cannot be written. According to conventions of the DEC-Alpha processor family the base address *base* is stored in register $R30$. Hence this register is also not assigned. Suppose we want to generate code for the instruction $addr_{0x001c} :=_I ld_I(addr_{0x001c}) +_I int_{0x0001}$. Fig. 6 shows a possible term-rewrite. The first column contains the actual term to be rewritten, the second column contains the rule that is applied, the third column shows the matching substitutions for the LHS of the rule, the fourth column shows a register assignment, and the last column shows the corresponding code generated. Reading the last column from top to bottom yields the machine instruction sequence implementing the intermediate language instruction. For any of the applied rules there are alternatives: Instead of applying rule (11) rule (7) is also applicable for loading the address constant in a register. Then, rule (10) has to be applied later. Similarly, rule (7) could be applied instead of rule (3). The final rule application would then be rule (1). Instead of applying rule (15) at the second step, one could have applied rule (5) to load the integer constant 1 to a register and then apply rule (13).

One property is that the applied rules overlap in exactly one node of the instruction tree, cf. Fig. 7 for the term-rewrite in Fig. 6. The values of these nodes must be stored in registers. A generator for code selection computes for each instruction of the intermediate program such a rule cover, assigns registers to the overlapping nodes (these are used during term-rewriting for assigning registers to variables), and plans the order of rule applications. Then, the term-rewrite is actually executed as demonstrated by Example 3. Formally the register assignment *ra* is used as a substitution of the variables in the term-rewrite rule by registers.

4 Correctness of Program Transformations

The notion of compiler or translation correctness is often defined as refinement of programming language constructs. In particular each state transition defined by a source language concept (e.g. a conditional statement) must be implemented in exactly the same way by the target machine. However, this may forbid some global or interprocedural optimizations (although for the purpose of this paper it might be sufficient). The notion of compiler correctness took in *Verifix* longer than expected. A more extensive discussion can be found in [24,21].

Observable Behaviour. From a compiler user's viewpoint, only the input/output relation of the program is of interest. Each program has such an interaction with an environment which we call *observable behaviour*. In terms of ASMs the states are projected to the I/O-relevant dynamic functions, e.g. the input/output streams (*observable states*). The *observable behaviour* of a program consists only of the observable states and state transitions between them induced by the more fine semantics. It is an abstraction of the ASM semantics as state transition system and therefore also a state transition system. Compiler users usually only require that the target program preserves the observable behaviour of the source program.

Resource Limitations. Since usually machine resources are limited while it is easy to write e.g. Java programs that would consume more than 10TByte memory, the target programs may exceptionally stop because of memory overflow. We therefore came up with the following notion of correctness: Let τ be a program of the target language with the observable behaviour $(I, Q, \rightarrow)$ and σ be a program of the source language with observable behaviour $(I', Q', \rightarrow')$. τ *preserves the observable behaviour* of σ *up to resource limitations* iff there is a relation $\phi \subseteq Q \times Q'$ such that for any finite or infinite sequence $q_0 \rightarrow q_1 \rightarrow \cdots$ of τ with $q_0 \in I, q_1, q_2, \ldots \in Q$ there is a finite or infinite sequence of states $q'_0 \rightarrow q'_1 \rightarrow \cdots$ of σ with $q'_0 \in I'$ and $q_i \phi q'_i$ for all i except possibly for the last state (if the sequence of observable states of τ is finite). This means that τ halts with violation of resource limitations. Fig. 8 visualizes this definition.

The preservation of observable behaviour up to resource limitations is transitive and therefore can be applied stepwise for the different phases in a compiler. It might be even useful for the purpose of proving correctness to introduce new intermediate languages that are not used by a compiler. If we speak in the following about source and target language, this may be one intermediate language (before the transformation) and the next intermediate language (after the transformation).

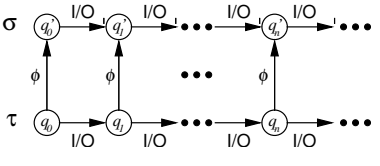

Fig. 8. Preservation of Observable Behaviour

Example 4. Consider the following language (called BBMIX) which is the union of the intermediate language in Example 1 and of basic block graphs of DEC-Alpha machine instructions (except jumps) which is obtained before assembling the binaries. The notion of programs, procedures, and basic blocks is analogous to Example 1. However, the set of instructions is the union of the set of intermediate language instructions (see Table 1) and the set of DEC-Alpha instructions except DEC-Alpha jump-instructions. Expressions are enriched by register access $rg(i)$ (abbreviated as Ri). A basic block may contain instruction sequences such as LDQ $R2, (28)R30; addr_{28} :=_I R2 +_I int_1$.

For the dynamic semantics, we extend expression evaluation by access to registers, i.e. $eval(Ri) = reg(i)$ and otherwise use all the state transition rules from the intermediate language (cf. Fig. 2) and the DEC-Alpha (except jumps, cf. Fig. 4) but with the *Proceed*-macro from the intermediate language. Note that except the expression evaluation for registers nothing need to be added for the definition of BBMIX. Anything else can be derived completely from the language definition for the intermediate language and the DEC-Alpha machine language. The intermediate language and the basic block graphs with DEC-Alpha machine instructions are now just sub-languages from BBMIX. The transformation rules in Fig. 4 are now program transformations within BBMIX. E.g. applying only Rule 11 to the instruction $addr_{28} :=_I ld_I(addr_{28}) +_I int_1$ would yield the instruction sequence LDQ $R2, (28)R30; addr_{28} :=_I R2 +_I int_1$. Therefore we can identify source and target language.

Remark 1. We were able to perform this process of language unification using ASM-semantics under rather general conditions, see [53]. It is independent on the concrete instruction sets of the intermediate language and the target language, it only requires a notion of expression in the intermediate language and the notion of registers in the target language. Note that source-to-target transformations and local optimizing transformations can be dealt within the same way.

Correctness of Program Transformations. A program transformation is *correct* iff any target program τ obtained by the program transformation from a source program σ preserves the observable behaviour of σ up to resource limitations. The correctness proof for program transformations follows the idea of simulation proofs similar to those in complexity and computability theory. One has to define a relation ρ between the states of programs of the target language and states of programs of the source language that is compatible to the relation ϕ on the observable behaviours of the target and source programs, respectively. This could be the same language as demonstrated by Example 4. The first simulation in Fig. 9 shows the conditions on ρ if the observable behaviour in the source and target program does not change (i.e. the diagrams must commute). The second simulation shows if there is exactly one observable state transition in the target and source program.

Local and Global Correctness. If a program transformation replaces a program fragment ψ' by another program fragment ψ then the state initial at ψ' and ψ and final at ψ' and ψ are in relation ρ, respectively. For a simulation proof

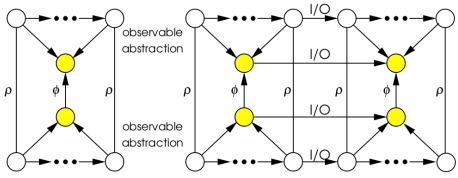

Fig. 9. Simulation Proofs

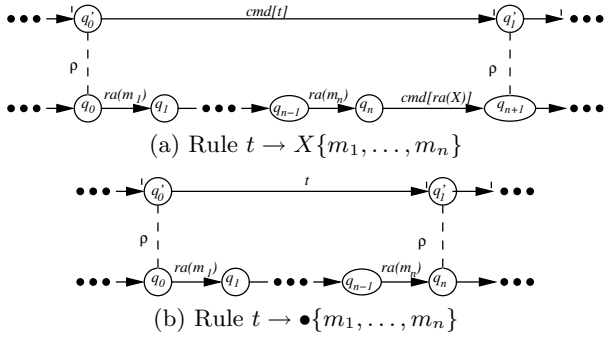

Fig. 10. Local Correctness of Term-Rewrite Rules w.r.t register assignment *ra*

two properties have to be shown: First one has to show that concrete program transformations are correct in the above sense (*local correctness*). Second, the single simulations stemming from program transformations must be sequentially composable (*global correctness*). We now apply these notions for code selection. In the following we assume that the intermediate language and the target language are united in the way as demonstrated by Example 4, for details see [53]. This has the advantage we can focus on the application of a single transformation rule. The complete simulation follows by induction on the number of applied transformation rules. Since we focus on the application of a single transformation rule, we have a mapping ψ from the addresses of the instruction in the source program and the addresses in the target program. We can now define the relation ρ: Let Q' and q be the sets of states before and after the single application of this transformation rule, respectively. Two states $q \in Q$, $q \in Q'$ are said to be *corresponding* iff $[\![f]\!]_q = [\![f]\!]_{q'}$ for all dynamic functions except for *ip* and registers containing dead values w.r.t. q'[5]. The idea behind corresponding states is that their relevant memory is isomorphic. We define $q\rho q'$ iff q and q' are corresponding states and $\psi([\![ip]\!]_{q'}) = [\![ip]\!]_q$.

We first define local correctness. Let Q' and Q be defined as above, and *ra* be the register assignment computed during the planning phase of the code selection. A term-rewrite rule of the form $t \to X\{m_1, \ldots, m_n\}$ is *locally correct* w.r.t. *ra* iff for all states $q_0, \ldots, q_n \in Q$ such that $q'_0 \models instr(ip) = cmd[t]$,

[5] A value is *dead* if it is not needed as operand by instructions executed later.

$q_0 \to q_1 \to \cdots \to q_{n-1} \to q_n$, $q_0 \models instr(ip) = ra(m_1), \ldots, q_{n-1} \models instr(ip) = ra(m_n)$, and $q_n \models instr(ip) = cmd[ra(X)]$ there is state $q_0' \in Q'$ such that $q_0 \rho q_0'$ and the following two conditions are satisfied:

i. $[\![eval(t)]\!]_{q_0'} = [\![reg(ra(X))]\!]_{q_n}$ and
ii. q_n and q_0' are corresponding states. Note that $reg(ra(X))$ must contain a dead value in q_0'.

This definition implies $q_{n+1} \rho q_1'$ since expression evaluation in the intermediate language is free of side-effects.

A term-rewrite rule $t \to \bullet\{m_1, \ldots, m_n\}$ is *locally correct* w.r.t. *ra* iff for all states $q_0, \ldots, q_n \in Q$ such that $q_0 \to q_1 \to \cdots \to q_{n-1} \to q_n$ where $q_0 \models instr(ip) = ra(m_1), \ldots, q_{n-1} \models instr(ip) = ra(m_n)$ there is a state q_0' such that $q_0 \rho q_0'$ and the following two conditions are satisfied:

iii. q_n and q_1' are corresponding states.
iv. $q_0' \models instr(ip) = t$ and $[\![instr(ip)]\!]_{q_n} = [\![instr(ip)]\!]_{q_1'}$.

Thus it holds $q_n \rho q_1'$, cf. Fig. 10(b).

Global correctness can be proven under rather general conditions. In [53] we have proven that the following theorem holds for any transformation from basic block oriented intermediate languages with expressions to register machines:

Theorem 1 (Global Correctness of Simulation). *Let $\mathcal{T}$ be a set of term-rewrite rules specifying the transformation in a code-selection, σ be a source program annotated with register assignment ra, a rule cover, and a schedule on the order of applying rules, and τ be the target program obtained by executing the program transformations according to the schedule. If each rule in $\mathcal{T}$ is locally correct and for any applied rule $t \to X\{m_1, \ldots, m_n\}$, register $ra(X)$ (the register assigned to hold the value for t) does contain a live value, then τ preserves the observable behaviour of σ.*

The proof has two stages. First, it is by induction on the number of applied term-rewrite rules as described above. The corresponding simulation is then shown by induction on the number of executions of the program fragment the transformation was applied to (actually more recent work on infinite state sequences indicates that it is indeed co-induction, cf. [22]). The inductive step requires the local correctness conditions for term-rewrite rules.

The global correctness theorem is rather independent of the languages and only requires the notion of expressions and instruction pointers stemming from basic-block graph based intermediate languages, and the notion of register stemming from register-based target processor architectures, i.e. the same requirements as for code selection by term-rewriting. The precondition on live values can be analyzed upon compilation and is therefore part of the verification of the back-end implementation. It is a good candidate for program checking approaches [15,17]. However, proving the local correctness conditions is language-dependent and should be done for each compiler.

5 Proof Strategies for Correctness of Code Selection

Typical compiler-backends often require several thousand term-rewrite rules. The correctness of term-rewrite rules therefore cannot be proven manually. The main idea to prove mechanically correctness of term-rewrite rules is to symbolically execute the LHS of a term-rewrite rule and the code to be generated from it. For each of the two classes of term-rewrite rules there is one simple proof strategy.

Consider first term-rewrite rules of the form $t \to X\{m_1, \ldots, m_n\}$. We have basically to ensure that Condition (i) is satisfied and that no other register changes its content. The variables in the term-rewrite rule stand for registers containing some live values. We therefore use each of these variable names as symbolic register names. The expression evaluation $eval(t)$ is executed symbolically with these symbolic register names using the definition of Fig. 2 extended by reading register access as described by Example 4. Furthermore, the instruction sequence $m_1, \ldots, m_n$ is also evaluated symbolically using the state transition rules of Fig. 4. Then, the result will be checked whether $reg(X)$ contains the result of the symbolic evaluation of $eval(t)$ and nothing else (except the instruction pointer) changed. The requirement on the instruction pointer is satisfied as long as the machine instruction sequence does not contain jump instruction.

Example 5. Consider Rule 6. The symbolic evaluation of a 32-bit integer constant yields $eval(int_{c32}) = SExt_{32}(c32)$. Suppose that $c32 = [b_{31}, \ldots, b_0]$. Then $SExt_{32}(c32) = [\underbrace{b_{31}, \ldots, b_{31}}_{32 \text{ times}}, b_{31}, \ldots, b_0]$. Symbolic execution of the first instruction $\mathsf{LDA}\ X, (c32.L)R31$ yields

$$\begin{aligned} reg(X) &= reg(R31) \oplus SExt_{16}(c32.L) \\ &= 0 \oplus SExt_{16}([b_{15}, \ldots, b_0]) \\ &= [\underbrace{b_{15}, \ldots, b_{15}}_{48 \text{ times}}, b_{15}, \ldots, b_0]) \end{aligned}$$

Then, the instruction $\mathsf{ZAP}\ X, \#fc, X$ is executed:

$$\begin{aligned} reg(X) &= zerobytes(reg(X), 11111100) \\ &= zerobytes([\underbrace{b_{15}, \ldots, b_{15}}_{48 \text{ times}}, b_{15}, \ldots, b_0], 11111100) \\ &= [\underbrace{0, \ldots, 0}_{48 \text{ times}}, b_{15}, \ldots, b_0] \end{aligned}$$

Finally, $\mathsf{LDAH}\ X, (c32.H)X$ is executed:

$$\begin{aligned} reg(X) &= reg(X) \oplus LogShift(SExt_{16}(c32.H)), 16) \\ &= [\underbrace{0, \ldots, 0}_{48 \text{ times}}, b_{15}, \ldots, b_0] \oplus LogShift([\underbrace{b_{31}, \ldots, b_{31}}_{48 \text{ times}}, b_{31}, \ldots, b_{16}], 16) \\ &= [\underbrace{0, \ldots, 0}_{48 \text{ times}}, b_{15}, \ldots, b_0] \oplus [\underbrace{b_{31}, \ldots, b_{31}}_{32 \text{ times}}, b_{31}, \ldots, b_{16}, \underbrace{0, \ldots, 0}_{16 \text{ times}}] \\ &= [\underbrace{b_{31}, \ldots, b_{31}}_{32 \text{ times}}, b_{31}, \ldots, b_0] \end{aligned}$$

All other dynamic functions except *ip* do not change, because the transition rules of ASMs explicitely specify state changes. In particular each dynamic function that is not updated remains unchanged.

This proof strategy always works for this kind of transformation rules since in a term-rewrite rule $t \to X\{m_1, \ldots, m_n\}$ the term t is just an expression and therefore does not cause side-effects.

Now, we consider term-rewrite rules of the form $t \to \bullet\{m_1, \ldots, m_n\}$. Here, a similar strategy as above is used. The difference is that the execution of t causes state changes (in particular in the memory) and these state changes must be the same as caused by the machine instruction sequence $m_1, \ldots, m_n$. Therefore, we symbolically execute the state transition caused by t using the state transition rules of Fig. 2 and symbolically execute the machine instruction sequence using the state transition rules of Fig. 4 and the fact $base = reg(R30)$, i.e., the base address is stored in register $R30$. We use symbolic register addresses and compare the resulting state changes according to condition (iii).

Example 6. Consider Rule 3. According to the state transitions in Fig. 2, after execution of $addr_{c16} := Y$, we obtain the only state change

$$\begin{aligned} mem(eval(addr_{c16})) := eval(rg(Y)) &= mem(base \oplus SExt_{16}(c16)) := reg(Y) \\ &= mem(reg(R30) \oplus SExt_{16}(c16)) := reg(Y) \end{aligned}$$

According to the state transitions of Fig. 4, executing $\mathsf{STQ}\ Y, (c16)R30$ would yield the state transition: $mem(reg(R30) \oplus SExt_{16}(c16)) := reg(Y)$ As one can see, these are exactly the same updates which are executed.

If more than one machine instruction is executed then the updates stemming from the machine instructions are composed. Again, this proof strategy always works for term-rewrite rules of the form $t \to \bullet\{m_1, \ldots, m_n\}$ because here t is an instruction and it is the state transitions that are of interest.

These two proof strategies can easily be implemented. The implementation can be parameterized with the semantics of the intermediate and the target language. The size of the equalities to be proven could explode exponentially in n, the number of machine-instructions to be generated. However, this number is usually rather small.

6 Assembly

For a complete back-end remains to discuss assembly. The binary has to be mapped into the linear memory of the target machine. Thus, each label of a basic block is mapped to the address of the first instruction of the basic block and each jump-instruction has as an argument the address of the basic block associated with the jump target. There are three possible cases to map jumps: First, two basic blocks bb_1 and bb_2 are mapped consecutively and bb_2 is the single successor of bb_1. Then, no jump is necessary. If the jump instruction and the jump target are close enough, a relative jump can be made, i.e. the relative address of the jump target is directly encoded in the jump instruction. Otherwise, the jump target has to be loaded as a constant into a register. The registers used for loading this constant must not contain live values. Note that all registers assigned by local register assignment in the code selection phase satisfy this property. Hence registers with these properties can be determined at compile time.

$jmp(L) \rightarrow \bullet\ \{\}$	if $l = current \oplus_Q 4$	Rule 17
$jmp(L) \rightarrow \bullet\ \{\mathsf{BR}\ Ri, \#l_{41} \cdots l_{61}\}$	if $-2^{22} \leq l < 2^{22}$	Rule 18
$jmp(L) \rightarrow \bullet\ \{\mathsf{LDA}\ Ri, (l.LL)R31$ $\mathsf{ZAP}\ Ri, \#fc, Ri$ $\mathsf{LDAH}\ Ri, (l.LH)Ri$ $\mathsf{BR}\ Rj, \#000001$ $\mathsf{ADD}\ Ri, Rj, Ri$ $\mathsf{ADD}\ Ri, \#08, Ri$ $\mathsf{JMP}\ Rj, Ri\}$	if $-2^{31} \leq l < -2^{22}$ or $2^{22} \leq l < 2^{31}$	Rule 19
$jmp(L) \rightarrow \bullet\ \{\mathsf{LDA}\ Ri, (l.HH)R31$ $\mathsf{SLL}\ Ri, \#10, Ri$ $\mathsf{LDA}\ Ri, (l.HL)Ri$ $\mathsf{SLL}\ Ri, \#10, Ri$ $\mathsf{LDA}\ Ri, (l.LH)Ri$ $\mathsf{SLL}\ Ri, \#10, Ri$ $\mathsf{LDA}\ Ri, (l.LL)Ri$ $\mathsf{BR}\ Rj, \#000001$ $\mathsf{ADD}\ Ri, Rj, Ri$ $\mathsf{ADD}\ Ri, \#08, Ri$ $\mathsf{JMP}\ Rj, Ri\}$	if $l < -2^{31}$ or $l \geq 2^{31}$	Rule 20

where $current = addrbb(labjmp(L)) \oplus_Q lenbb(lab(jmp(L)))$, $l = addrbb(L) - current$, $l.LL$, $l.LH$, $l.HL$, and $l.HH$ are the 1st, 2nd, 3rd, and 4th sixteen bits of l, and Ri. Rj are registers which don't contain live values

Fig. 11. Mapping of Jumps

Example 7. We have to map basic block graphs consisting of DEC-Alpha machine instructions and symbolic jumps to a linearized assembly program. The mapping $addrbb : LABEL \rightarrow QUAD$ maps each label of a basic block to a relative address. The mapping $lenbb : LABEL \rightarrow QUAD$ maps each basic block (identified by its unique symbolic label) to its length in the binary code. These mappings have to be computed by a compiler. Note that this implies that the address of the jump instruction *current* and the relative distance l to the jump target can be computed at compile time[6]. The transformations in Fig. 11 are used for mapping unconditional jumps. The transformations for conditional jumps are analogous. The first transformation is used if the successor block is mapped consecutively, the second is used if the relative distance to the jump target can be encoded as a 21-bit relative address. The last two transformations load the jump target directly into register Ri. Note that the registers required for this transformation must be free. The instruction $\mathsf{BR}\ Rj, \#000001$ is required to load $pc \oplus_Q 4$ into register R_j. The two ADD-instructions adjust the correct address of the jump target since it must be relative to the address of the last instruction of the basic block. Note that this address is given by *current*. It is not difficult to prove that these transformations are locally correct. The proofs follow the same strategy as in Fig. 10(b).

Similar to the code selection phase, the compiler performs a planning phase where it computes *addrbb*, for each basic block the transformations applied to its jump instructions, and a function $jumps : LABEL \rightarrow QUAD$ that computes

[6] $lab(jmp(L))$ is the label of the basic block that contains the jump instruction $jmp(L)$ where the transformation rule is applied.

the size of these jump instructions. For the example of the DEC-Alpha processor, it holds:

$$jumps(L) = \begin{cases} -4 & \text{if Rule 17 is applied to the jump in } lab^{-1}(L) \\ 0 & \text{if Rule 18 is applied to the jump in } lab^{-1}(L) \\ 24 & \text{if Rule 19 is applied to the jump in } lab^{-1}(L) \\ 40 & \text{if Rule 20 is applied to the jump in } lab^{-1}(L) \end{cases}$$

Note that $lenbb(L) = length(lab^{-1}(L)) \otimes_Q 4 \oplus jumps(L)$ since each instruction requires 4 bytes space (*length* denotes the length of a sequence). The total size of the code is $lenbb(L_0) + \cdots + lenbb(L_m)$ where $L_0, \ldots, L_n$ are the labels of all basic blocks in the program. The assembly phase might optimize this total code size. Alternatively, other criteria might be used.

For the correctness of the assembly phase, it is not hard to see that it is sufficient to prove the conditions in Fig. 12. The first condition states that instructions of a basic block are mapped consecutively without gaps in the same order as in the basic block. The second condition states that two basic blocks do not overlap. The last condition states that the transformation rules of Fig. 11 are correctly applied and that $jumps(L)$ is large enough to store the jump instructions of the basic block with label L. Note that all conditions of the transformation rules can be checked at compile time. Together with the local correctness of the transformations for the jumps this implies by induction on the number of state transitions the global correctness of the assembly phase.

The technique of program checking may be used for checking the conditions in Fig. 12. Note that checking these proof obligations except the last one is independent of the concrete target language because the functions $instr_\alpha$ *addrbb*, *lenbb*, *length*, and *jumps* must be computed for any register based target language and basic-block based intermediate language. Checking the last condition requires only the knowledge of the transformation rule to be applied and the size of an instruction (which is 4 in the example of DEC-Alpha). It is not hard to see that this checking algorithm requires time $O(n + k \log k)$ where n is the number of instructions in the target program and k is the number of basic blocks. Checking the first and the third condition in Fig. 12 requirerconstant time (for a single instruction), hence they require a total of time $O(n)$. For checking the second condition, the labels are sorted according to their addresses. Then, it is sufficient to check the second condition only for the consequetive labels.

7 Related Work

The kind of code generation discussed in Section 3 was first introduced as bottom-up rewrite systems (BURS) [40]. Several works improve this technique. Emmelmann implemented this technique in BEG [14,13] and adds algebraic identities and uses dynamic programming to find cost-optimal rule covers. [44] uses tree automata to execute the term-rewriting. This has the additional advantage that completeness of the specification w.r.t. the intermediate language can be

$$instr(L,i) = instr(addrbb(L) + i \cdot 4) \quad \text{for all labels } L,\ 0 \le i < length(lab^{-1}(L)) \text{ and } instr(L,i) \text{ is not a jump instruction}$$

$$addrbb(L_1) - addrbb(L_2) < lenbb(L_2) \quad \text{for all labels } L_1,\ L_2,\ L_1 \neq L_2$$

For each transformation rule $JMP(L) \rightarrow \bullet\{m_1, \ldots, m_n\}$ if $cond$ applied in a basic block with label L'

$$jumps(l) = (n-1) \cdot 4 \wedge cond \wedge instr(addrbb(L') + l \cdot 4) = m_1 \wedge \\ \wedge \cdots \wedge instr(addrbb(L') \oplus_Q (l + n - 1) \cdot 4) = m_n$$

$$\text{where } l = length(lab^{-1}(L')) - 1$$

Fig. 12. Proof Obligations for Correctness of Assembly Phase

decided efficiently. Nymeyer et. al. [38] use A^*-search in order to find rule covers. None of these works discussed the correctness of code generation.

Correctness of compilers was first considered in [31]. They discussed the compilation of arithmetic expressions. Samet used an approach that we now call translation validation [45,46,48,47,49]. There are a number of works using denotational semantics, e.g. [8,33,39,43,52]. Other works use the approach of refining language constructs, e. g. [6,7,9,27,32,34,50], or structural operational semantics, e. g. [11]. Often these works consider single phases of a compiler. E.g. [50] discusses intermediate language generation. The code selection phase for generating binary code is often not considered in works on compiler correctness. [32] discusses the compilation of a stack-based intermediate language into a register based assembly language. [34,27] consider the Transputer as target-machines. Each of these works check transformations in hand-written compiler back-ends. They don't discuss correctness of transformation rules used by generated term-rewrite based compiler back-ends.

Program checking in code generators was independently developed from us in the area of safety-critical systems [41] and is called *translation validation* by them. The difference to general compilers is that their target code has a special form and it mainly consists of an implementation of a finite state machine. Zuck et. al. extended the ideas to validate certain optimizing transformations [54,56,55,2,23]. Necula uses a similar approach for checking local optimizations in basic blocks [36]. Glesner and Blech apply translation validation to constant-folding [18]. Glesner et. al. use translation validation for correctness for the lexical analysis of the GNU C compiler [19] and generalize the approach of [15] to code generators for embedded systems [20]. However, translation validation does not help to identify erroneous transformation rules - it just tells that something in the compilation went wrong.

In contrast to our work, works on verifying compiler optimization focuses on specific parts of a compiler. Blech and Glesner prove the correctness of some optimizations using Isabelle/HOL [4,3]. Lacey et. al. use temporal logic for this purpose [28,29]. These works have in common that they do not change the language level. Strecker shows the correctness of transformations from a subset of Java (μJava) to Java Byte Code using Isabelle/HOL [51]. Schmid et. al. showed the proof for Java except threads [50] but used a paper and pencil approach. Blech, Glesner et.al. show the correctness of translations of SSA-intermediate languages to machine languages[5]. Poetzsch-Heffter and Gawkowski propose a

similar approach for C[42]. Both approaches use Isabelle/HOL. They assume that there is one-to-one correspondence between machine operations and intermediate language operations. Dold, v. Henke and Goerigk developed in *Verifix* a completely verified compiler (in binary code) for a LISP subset[12]. The proofs were checked using PVS. Leinenbach, Paul, and Petrova developed in the framework of the *VeriSoft*-project a verified macro-expansion based compiler for a Pascal/C-subset using Isabelle/HOL[30]. A final remark: Proof Carrying Code [35] and Certifying Compilers [37,10] check necessary conditions for the correctness of compilation while *Verifix* and translation validation approaches check sufficient conditions.

8 Conclusions

We have shown how the transformations in compiler back-ends can be completely verified. It requires abstract state machine specifications for the semantics of the intermediate and target languages, respectively. The local correctness of the transformation rules can then be mechanically verified by using two proof strategies that suffice if the transformations for code selection and assembly are given as term-rewrite rules. The composition as simulation proofs is guaranteed under rather general requirements to intermediate and target languages, respectively. However, for the code selection the compiler has to check whether no register is written that contains a live value, i.e., a value that is still required. Having annotations to the intermediate program giving a rule cover for each instruction, a register assignment w.r.t. a rule cover, and a schedule specifying the order of application of term-rewrite rules, this can easily be checked independently of the compiler. Similarly, the compiler has to check the conditions in Fig. 12 for the assembly phase. This suggests to use techniques of program checking for that purpose. If this program checker and the module actually performing the term-rewriting is verified, it is guaranteed that any code generated by a compiler actually preserves the observable behaviour of the intermediate program.

It should be noted that the proof strategies for term-rewrite rules also work for local optimizations. However, for global optimizations such e.g. code motion, it does not work since they cannot be expressed as local transformation rules on trees. Instead, graph-rewrite rules should be used. Adding edges stemming from data-flow information might result in local graph transformations as e.g. be used in optimizer generators [1]. The same idea could be applied when instruction-scheduling techniques are applied. For pipelined architecturs and instruction-level parallelism SSA-graphs are the more suitable intermediate representation. First steps towards this direction are made[5]. However, it does not yet cover the full power of instruction scheduling approaches. This is subject to further research.

The strength of formal approaches was demonstrated by finding a serious bug in the specification of DEC-Alpha back-end developed by a student project. It was an erroneous version of the transformation for loading large constants discussed in Section 5. It was accidently overseen that loading 16-bit constants into register automatically sign-extends the constant. If correctness proofs fail, error messages should provide claims that have to be proven for ensuring local

correctness of transformation rules. In the above case, this would ideally produce an error message requiring to prove that the bit 15 of a constant must be 0. This is a precise hint on what went wrong. From a practical viewpoint the use of formal methods in *Verifix* turned out to be successfull.

The lessons we learned about the use of the formal methods is that they should satisfy several requirements to be successfull in practice:

- A formal method shouldn't restrict the problem to be solved in any way. E.g. in *Verifix* we ruled out denotational semantics for language semantics since it seems that there are some requirements on compositionality.
- A formal method should take into account practical requirements. E.g. the notion of correctness in *Verifix* had to take into account resource limitations because of practical needs. This notion deviates from that in more theoretical approaches.
- A formal method should not restrict in any way design decisions for systems to be build. Otherwise, it won't be accepted by practicioners. E.g., in *Verifix* we stressed that by keeping the well-established architecture of compilers.
- Tool support is necessary because of the size and complexity of the proofs to be performed. Their underlying formal method should support tools to produce helpful error messages. In *Verifix* we used PVS for that purpose.

In particular the last issue is important for an increasing acceptance of formal methods by practicioners.

Acknowledgements. I thank the two anonymous referees for their helpful comments. I'm grateful to all colleagues of the Verifix project in Karlsruhe, Kiel, and Ulm for the inspiring and fruitful discussions.

References

1. U. Assmann. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems*, 22(4):583–637, 2000.
2. C. Barrett, B. Goldberg, and L. Zuck. Run-time validation of speculative optimizations using CVC. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.
3. J. O. Blech, L. Gesellensetter, and S. Glesner. Formal verification of dead code elimination in Isabelle/HOL. In *3rd IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society Press, 2005. to appear.
4. J. O. Blech and S. Glesner. A formal correctness proof for code generation from SSA form in Isabelle/HOL. In *Informatik 2004*, number P-51 in Lecture Notes in Informatics, pages 449–458. Springer, 2004. Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft fr Informatik.
5. J. O. Blech, S. Glesner, J. Leitner, and S. Mlling. Optimizing code generation from SSA form: A comparison between two formal correctness proofs in Isabelle/HOL. *Electronic Notes in Theoretical Computer Science*, to appear, 2005. Proceedings of the 4th COCV-Workshop (Compiler Optimization meets Compiler Verification).
6. E. Brger and I. Durdanovic. Correctness of compiling occam to transputer. *The Computer Journal*, 39(1):52–92, 1996.

7. E. Brger, I. Durdanovic, and D. 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.
8. 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.
9. 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, 1992.
10. C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107. ACM Press, 2000.
11. S. Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, Universität Saarbrücken, 1996.
12. A. Dold, F. W. von Henke, and W. Goerigk. A completely verified realistic bootstrap compiler. *International Journal on Foundations of Computer Science*, 14(4):659–680, 2003.
13. H. Emmelmann. *Codeselektion mit regulär gesteuerter Termersetzung*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, GMD-Bericht 241, Oldenbourg-Verlag, 1994.
14. H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG – a Generator for Efficient Back Ends. In *Proceedings of the Sigplan '89 Conference on Programming Language Design and Implementation*, June 1989.
15. T. Gaul, A. Heberle, W. Zimmermann, and W. Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. In A. Pnueli and Paolo Traverso, editors, *Proceedings of RTRV '99: Workshop on Runtime Result Verification*, Trento, Italy, 1999.
16. T.S. Gaul. An Abstract State Machine Specification of the DEC-Alpha Processor Family. Verifix Working Paper [Verifix/UKA/4], University of Karlsruhe, 1995.
17. S. Glesner. Using program checking to ensure correctness of compiler implementations. *Journal of Universal Computer Science*, 9(3):191–222, 2003. Special Issue on Compiler Optimization meets Compiler Verification.
18. S. Glesner and J.-O. Blech. Classifying and formally verifying. In *2nd Workshop on Compiler Optimization meets Compiler Verification COCV2003*, volume 82 of *Electronic Notes in Theoretical Computer Science*, 2003.
19. S. Glesner, S. Forster, and M. Jäger. A program result checker for the lexical analysis of the gnu c compiler. In *3rd International Workshop on Compiler Optimization meets Compiler Verification COCV2004*, Electronic Notes in Theoretical Computer Science, 2004.
20. S. Glesner, R. Geiß, and B. Bösler. Verified code generation for embedded systems. In *1st Workshop on Compiler Optimization meets Compiler Verification COCV2002*, volume 65 of *Electronic Notes in Theoretical Computer Science*, 2002.
21. S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender bersetzer. *IT – Information Technology*, 46(5):265–276, 2004.
22. S. Glesner and W. Zimmermann. Structural Simulation Proofs based on ASMs even for Non-Terminating Programs. In *Proceedings of the ASM-Workshop, Eight International Conference on Computer Aided Systems Theory EUROCAST 2001*, Feb 2001.

23. B. Goldberg, L. Zuck, and C. Barrett. Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1), 2005.
24. G. Goos and W. Zimmermann. Verification of compilers. In B. Steffen E.-R. Olderog, editor, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 201–230. Springer, 1999.
25. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Brger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
26. Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department, 1997.
27. C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.
28. D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 283–294. Association of Computing Machinery, 2002.
29. D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order and Symbolic Computation*, 17(3):173–206, 2004.
30. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society Press, 2005. to appear.
31. 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.
32. J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Kluwer Academic Press, Dordrecht, The Netherlands, 1996.
33. 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.
34. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, New York, 1997.
35. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM Press, 1997.
36. G. C. Necula. Translation validation for an optimizing compiler. In *PLDI'00: SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–95. ACM, 2000.
37. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.
38. A. Nymeyer, J.-P. Katoen, Y. Westra, and H. Ablas. Codegeneration = A^*+BURS. In *Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 1996.
39. J. Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *IEEE International Conference on Computer Languages*, 1992.

40. E. Pelegr-Llopart and S.L.Graham:. Optimal code generation for expression trees: An application of BURS theory. In *Principle of Programming Languages POPL'88*, pages 294–308. ACM, 1988.
41. A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation for synchronous languages. *Lecture Notes in Computer Science*, 1443, 1998.
42. A. Poetzsch-Heffter and M. Gawkowski. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1), 2005.
43. W. Polak. Compiler specification and verification. In J. Hartmanis G. Goos, editor, *Lecture Notes in Computer Science*, volume 124 of *Lecture Notes in Computer Science*. Springer, 1981.
44. T. A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, 1995.
45. H. Samet. *Automatically proving the correctness of translations involving optimized code.* PhD thesis, 1975.
46. H. Samet. Compiler testing via symbolic interpretation. In *ACM 76: Proceedings of the annual conference*, pages 492–497, New York, NY, USA, 1976. ACM Press.
47. H. Samet. A machine description facility for compiler testing. *IEEE Transactions on Software Engineering*, 3(5):343–351, 1977.
48. H. Samet. A normal form for compiler testing. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 155–162, 1977.
49. H. Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, 1978.
50. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine.* Springer, 2001.
51. M. Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.
52. M. Wand. A semantic prototyping system. *SIGPLAN Notices*, 19(6):213–221, June 1984. SIGPLAN 84 Symposium On Compiler Construction.
53. W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM-Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.
54. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A Translation Validator for Optimizing Compilers. In J. Knoop and W. Zimmermann, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.
55. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.
56. L. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation and run-time validation of optimized code. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002.