

An ASM Semantics for SSA Intermediate Representations

Sabine Glesner

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe, 76128 Karlsruhe, Germany
<http://www.info.uni-karlsruhe.de/~glesner>

Abstract. Static single assignment (SSA) form is the intermediate representation of choice in modern optimizing compilers for which no formal semantics has been stated yet. To prove such compilers correct, a formal semantics of SSA representations is necessary. In this paper, we show that abstract state machines (ASMs) are able to capture the imperative as well as the data flow-driven and therefore non-deterministic aspects of SSA representations in a simple and elegant way. Furthermore, we demonstrate that correctness of code generation can be verified based on this ASM semantics by proving the correctness of a simple code generation algorithm.

1 Introduction

Because static single assignment (SSA) representations allow for the explicit representation of data flow as well as control flow dependencies, they are the preferred intermediate representation in modern optimizing compilers. Optimizations in compilers are typically the most error-prone parts, cf. e.g. [New01]. Such errors can only be eliminated if the applied optimizations are verified. To prove them correct, we first of all need a formal semantics of the employed intermediate representation. In this paper, we state a formal semantics for SSA representations. We require it to capture the imperative nature of SSA representations as well as their non-deterministic data flow driven character equally well. Furthermore, it should be applicable in correctness proofs for optimizing compilers.

Our semantics for SSA representations is formalized as an abstract state machine (ASM) [Gur95]. Each state during computation characterizes the current basic block. We capture the imperative, state-based part of SSA computations by transition rules which transfer control flow from one basic block to its successor basic block. Within basic blocks, SSA computations are purely data flow driven. We model these computations by transition rules which are non-deterministic in the sense that at a given point during the run of the ASM, more than one rule might be applicable. Our specification of SSA semantics is well-suited to prove the correctness of code generation algorithms. In this paper, we prove the correctness of a relatively simple machine code generation algorithm. Thereby we prove that a generated deterministic machine program preserves the data flow dependencies of the source SSA program by showing that the sequence of applied transition rules during the execution of the machine program corresponds

to one possible sequence of transition rules in the non-deterministic SSA semantics. Furthermore, we point out how this proof can be extended to capture also more complex optimization strategies during code generation.

This paper is structured as follows: First we introduce SSA representations in section 2. Then we describe how ASMs are typically used in the specification of the formal semantics of programming languages, cf. section 3. Afterwards, in section 4, we state our formal semantics for SSA representations based on ASMs. In section 5, we demonstrate that this formal semantics is a well-suited basis for correctness proofs of optimizing compilers. We complete this paper with a discussion of related work in section 6 and the conclusions in section 7.

2 SSA Intermediate Representations

Static single assignment (SSA) form has become the preferred intermediate program representation for handling all kinds of program analyses and optimizing program transformations prior to code generation [CFR⁺91]. Its main merits comprise the explicit representation of def-use-chains and, based on them, the ease by which further dataflow information can be derived.

By definition SSA-form requires that a program and in particular each basic block¹ is represented as a directed graph of elementary operations (jump/branch, memory read/write, arithmetic operations on data) such that each "variable" is assigned exactly once in the program text. Only references to such variables may appear as operands in operations. Thus, an operand explicitly indicates the data dependency to its point of origin. The directed graph of an SSA-representation is an overlay of the control flow and the data flow graph of the program. A control node may depend on a value which forces control to conditionally follow a selected path. Each basic block has one or more such control nodes as its predecessor. At entry to a basic block, ϕ nodes, $x = \phi(x_1, \dots, x_n)$, represent the unique value assigned to variable x . This value is a selection among the values x_1, \dots, x_n where x_i represents the value of x defined on the control path through the i -th predecessor of the basic block. n

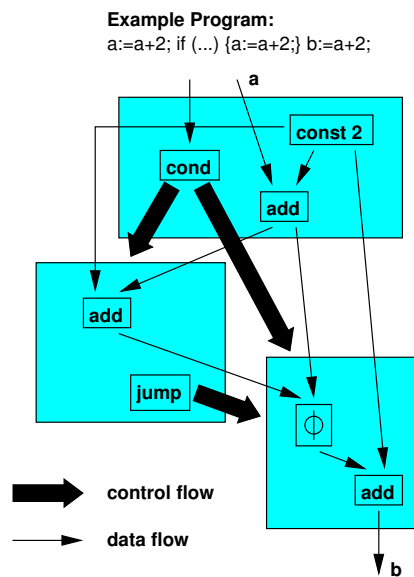


Fig. 1. SSA Representation

¹ A program is divided into basic blocks by determining maximal sequences of instructions that can be entered only at their first and exited from their last instruction.

is the number of predecessors of the basic block. Programs can easily be transformed into SSA representation, cf. [Muc97], e.g. during a tree walk through the attributed syntax tree. The standard transformation algorithm subscript each variable. At join points, ϕ nodes sort out multiple assignments to a variable which correspond to different control flows through the program.

As example, figure 1 shows the SSA representation for the program fragment:

```
a := a+2; if(..) { a := a+2; } b := a+2;
```

In the first basic block, the constant 2 is added to a. Then the *cond* node passes control flow to the ‘then’ or to the ‘next’ *block*, depending on the result of the comparison. In the ‘then’ *block*, the constant 2 is added to the result of the previous *add* node. In the ‘next’ *block*, the ϕ node chooses which reachable definition of variable ‘a’ to use, the one before the if statement or the one of the ‘then’ *block*. The names of variables do not appear in the SSA form. Since each variable is assigned statically only once, variables are identified with their value.

SSA representations describe imperative, i.e. state-based computations. A virtual machine for SSA representations starts execution with the first basic block of a given program. After execution of the current basic block, control flow is transferred to the uniquely defined subsequent basic block. Hence, the current state is characterized by the current basic block and by the outcomes of the operations contained in the previously executed basic blocks.

Memory accesses need special treatment. In the functional store approach [Ste95], memory read/write nodes may be considered as accesses to fields of a global state variable *memory*. A memory write access modifies this global variable *memory* and requires that the outcome of this write operation yields a new (subscripted) version of the *memory* variable. These duplications of the *memory* variable are the reason for inefficiencies in practical data flow analyses. As a solution, one might try to determine which memory accesses address overlapping memory areas and thus are truly dependent on each other and which address independent parts with no data dependencies. For the purpose pursued in this paper, these considerations are irrelevant. It is our goal to define a formal semantics for SSA representations. The same semantic description can be used for accesses to only a single as well as for accesses to several independent memories.

3 Abstract State Machines (ASMs)

ASMs [Gur95] are a general computation model to describe all kinds of computations as e.g. programming languages, hardware architectures, distributed systems, or real-time protocols. Especially for programming languages, many specifications of existing languages exist (e.g. C [GH93], C++ [Wal95], Java [SSB01], and SDL [EGGP00]). In this section, we summarize ASMs with respect to this particular use in the specification of programming languages.

3.1 Semantics of Programming Languages

The semantics of programming languages is in general compositional. Given a program in form of its abstract syntax tree, the semantics of each node can be

defined directly given its immediate successors. Nevertheless, certain constructs in programming languages exhibit a semantics which is inherently not compositional. E.g., `goto`-statements may leave a program part and go to some other place which cannot be described via the predecessor or successor relation in abstract syntax trees. ASM semantics is able to describe such non-compositional semantics. Therefore, *continuations* are defined. They describe where to proceed with the computation. If the control flow branches at a node, then several such continuations are defined, each describing the succeeding computation depending on the branch direction. The abstract syntax tree together with these continuations is the basis to define the semantics of programming languages.

SSA representations define the control flow by explicitly specifying the continuations, i.e. the control flow from one basic block to its successor basic block. Hence, ASMs are a well-suited framework to define a formal SSA semantics, cf. section 4, because they can utilize the explicitly stated continuations directly.

3.2 ASM Semantics for Programming Languages

Abstract state machines describe the semantics of programming languages operationally as state transition systems based on the abstract syntax trees of programs. Part of the current state is the current task, a pointer to the node in the abstract syntax tree currently executed. During program execution, states are transformed into new states, thereby also updating the pointer to the current task. States are regarded as algebras over a given signature. Each n-ary function symbol is interpreted with an n-ary mapping. During a state transition, the interpretation \mathcal{I} of some of the function symbols may change. E.g., if a function symbol S specifies the state of memory, then a variable assignment $\mathbf{x} := \mathbf{v}$ changes the interpretation $\mathcal{I}(S)$ of the function symbol S for argument \mathbf{x} : $\mathcal{I}(S(\mathbf{x})) := \mathcal{I}(\mathbf{v})$ holds in the new state. In general, an ASM consists of four components $(\Sigma \cup \Delta, \mathcal{A}, \text{Init}, \text{Trans})$: The signature is composed of two disjoint sorted signatures, the signature of the *static functions* Σ and the signature of the *dynamic functions* Δ . \mathcal{A} is the *static algebra*, an order-sorted Σ -algebra interpreting the function symbols in Σ . *Init* is a set of equations over \mathcal{A} defining the *initial states* of \mathcal{A} . *Trans* is a set of *transition rules* for specifying the state transitions by defining or updating, resp., the interpretations of certain function values of functions in Δ . A $(\Sigma \cup \Delta)$ -algebra is a state of the ASM iff its restriction to Σ is the static algebra \mathcal{A} . If q is a state, $f \in \Delta$ is a function symbol, and t_i are terms over $\Sigma \cup \Delta$ with interpretations x_i in q , then update $f(t_1, \dots, t_n) := t_0$ defines the new interpretation of f in the succeeding state q' as

$$q' \models f(x_1, \dots, x_n) = \begin{cases} x_0 & \text{if for all } i, 0 \leq i \leq n, q \models t_i = x_i \\ f_q(x_1, \dots, x_n) & \text{otherwise} \end{cases}$$

A transition rule defines a set of updates which are executed in parallel:

if *Cond* **then** *Update*₁ ... *Update*_n **fi**

If $q \models \text{Cond} = \text{true}$ in state q , then *Update*₁ ... *Update*_n are executed in q .

When defining the semantics of programming languages, the abstract syntax tree is used as basis and meaning is attached to it. Thereby, it is assumed that the abstract syntax tree contains attributes defining all continuations, especially for the non-compositional changes of the control flow. The definition of the ASM models the program counter during program execution, thereby using the continuation attributes which might be split up according to the value of conditions (true case and false case). Here is the example of a transition rule defining the semantics of the while-loop, as stated in [GZ99]. CT ($CT =$ current task) is the abstract program counter, $CT.TT$ (true task) is the true-continuation attribute of CT and $CT.FT$ (false task) is the false-continuation attribute of CT .

```

if  $CT \in \textit{While}$  then
  if  $\textit{value}(CT.\textit{cond}) = \textit{true}$  then  $CT := CT.TT$ 
  else  $CT := CT.FT$  fi fi

```

The semantics of each program node is described by a finite set of transition rules. Typically the condition of such a transition rule specifies the nodes in the abstract syntax tree (**While**-nodes in our example) for which the transition rule is applicable. The transition rules define updates, thereby employing children nodes (in our example $CT.\textit{cond}$) and statically computed continuations ($CT.TT$ and $CT.FT$ in our above example). In the remainder of this paper, we show how the semantics of SSA representations can be specified with ASMs. Furthermore, we prove the correctness of code generation based on this specification.

4 An ASM Semantics for SSA Representations

In this section, we present an ASM semantics for SSA representations. We first show how the structure of SSA programs is defined. Then we proceed by defining the transition rules describing the dynamic behavior of SSA programs.

$\textit{belongs_to}$: Operations \rightarrow BasicBlocks
$\textit{Pred}_1, \textit{Pred}_2, \textit{Pred}_3$: Operations \rightarrow Operations
\textit{Preds}	: Operations \rightarrow List(Operations \times BasicBlocks)
\textit{select}	: List(Operations \times BasicBlocks) \times BasicBlocks \rightarrow Operations
\textit{kind}	: Operation \rightarrow $\{\phi, \textit{add}, \textit{AND}, \textit{read}, \textit{write}, \textit{jump}, \textit{branch}\}$
$\textit{initial_store}$: $\mathbb{N} \rightarrow \{\textit{undef}\}$ s. t. $\forall n \in \mathbb{N}. \textit{initial_store}(n) = \textit{undef}$

Fig. 2. Static Functions

Programs in SSA form are characterized by their basic blocks and by the connection between them. Each basic block contains ϕ operations, arithmetic and Boolean operations² as well as memory accesses. Moreover, a jump or branch

² For sake of simplicity, we only consider the “add” and “AND” operation as representatives of arithmetic and Boolean operations.

operation is contained which transfers control flow during computation to the succeeding basic block. Note that each operation belongs uniquely to one specific basic block. During computation, the SSA representation itself is not modified. Hence, we describe the static structure of SSA programs by the static functions of the ASM. They are listed in figure 2.

The function *belongs_to* specifies, for each operation $op \in \text{Operations}$, to which basic block $b \in \text{BasicBlocks}$ it belongs. SSA representations specify explicitly the data flow of a program. In our formalization, this is expressed with the functions $Pred_1$, $Pred_2$, and $Pred_3$. $Pred_i(op) = op'$ means that the result of op' is the i th input of operation op , i.e., op' is the i th predecessor of op . For the evaluation of ϕ nodes, we need to know what their predecessors are and to which basic blocks they belong. We represent this information by a list of pairs *Preds*. Each such pair consists of a predecessor operation and the basic block to which this predecessor operation belongs. The function *select* returns, for a given basic block b , the operation op such that (op, b) is an element of the list *Preds*. *kind* is a function which returns for each operation its name. We specify memory accesses according to the functional store approach [Ste95]. In particular, we model the store as a function which maps the natural numbers, i.e. the potentially infinitely many store cells, into the set of possible values, i.e. $(\text{Bool} \cup \mathbb{Z} \cup \{def, undef\})$. Initially, the content of each cell is *undef*, represented by the constant static function *initial_store*. When executing memory write operations, this assignment of values might be changed. Hence, in general, arbitrary functions mapping from \mathbb{N} into $(\text{Bool} \cup \mathbb{Z} \cup \{def, undef\})$ represent the value of memory during computation. In the functional store approach, the result of a memory write is the updated memory. Hence, the dynamic function *value* which assigns operations to their results maps also into this set of functions from \mathbb{N} into $(\text{Bool} \cup \mathbb{Z} \cup \{def, undef\})$, cf. figure 3.

$value : \text{Operations} \rightarrow \text{Value} \cup \{def, undef\} \cup (\text{Bool} \cup \mathbb{Z} \cup \{def, undef\})^{\mathbb{N}}$ $init : \text{Bool}$ $current_block, next_block, pred_block : \text{BasicBlocks}$

Fig. 3. Dynamic Functions

Basic blocks are evaluated by first evaluating the ϕ nodes, by then computing the arithmetic, Boolean and memory operations, and finally by computing the successor basic block. The dynamic constant *init* guides the evaluation of the ϕ nodes. During their evaluation, *init* is *true*, afterwards it is set to *false*. A state during computation is characterized by the current basic block *current_block*, by its predecessor block *pred_block*, and by its current evaluation phase represented by the value of *init*. Figure 3 summarizes the dynamic functions.

Remark: Note that the ϕ nodes of a block must be evaluated before its other operations. Otherwise, in case that a block is its own predecessor, the evaluation

of a succeeding operation might falsify the result of a delayed ϕ node evaluation.

Initially, the current block is *start_block*, a distinguished basic block representing the starting point of computation. Furthermore, there is another distinguished basic block *start_pred* representing the predecessor block of *start_block*. Since a program might have input values, we need to represent them in the SSA form. We do this with the block *start_pred* which contains constants representing

<pre> current_block = start_block pred_block = start_pred next_block = void $\forall op \notin start_pred.value(op) = undef$ init = true </pre>

Fig. 4. Initial States

the program inputs. The initial value for the successor block is *void* since no successor block has been computed yet. Also, no operation has been computed, hence all their values are set to *undef*. The dynamic constant function *init* is set to *true* initially because computation starts with the evaluation of the ϕ nodes.

Evaluation of a basic block starts with the evaluation of the ϕ nodes, cf. figure 5. All ϕ nodes are evaluated simultaneously, *init* is set to *false*, and the values of all other operations in the current basic block are reset to *undef*.

<pre> if <i>init</i> and <i>current_block</i> \neq <i>void</i> then forall $op \in \{op' \mid belongs_to(op') = current_block \wedge kind(op') = \phi\}$ do in parallel <i>value</i>(<i>op</i>) = <i>select</i>(<i>Preds</i>(<i>op</i>), <i>Pred</i>(<i>op</i>)) od; forall $op \in \{op' \mid belongs_to(op') = current_block \wedge kind(op') \neq \phi\}$ do in parallel <i>value</i>(<i>op</i>) = <i>undef</i> od; <i>init</i> := <i>false</i> fi </pre>

Fig. 5. Transition Rule for the Evaluation of ϕ nodes

After evaluating the ϕ nodes, *init* is *false*, and rules for arithmetic, Boolean, memory access as well as jump and branch operations might be applied whenever their conditions are fulfilled, i.e., whenever their input values are computed. In general, there might be several operations within one basic block which can be evaluated. This stems from the data-flow driven nature of computations within SSA basic blocks. The selection among the corresponding updates is non-deterministic. We formulate these non-deterministic updates with the choose constructor [Gur95]. In section 5 we show that this non-deterministic definition is sound in the sense that each evaluation order which fits to the acyclic data dependencies of SSA basic blocks yields the same result. Note that our notation “**choose** $Op \subseteq \{op \mid \text{some requirements}\}$ **in** $\mathcal{P}(\text{Operations})$ R_0 **endchoose**” in figure 6 ($\mathcal{P}(S)$ denotes the powerset of a given set S) is an abbreviation for the qualified choose construct: “**choose** v **in** U **satisfying** $g(v)$ R_0 **endchoose**”.

The rules for arithmetic and Boolean operations as well as for memory accesses follow the same schema: If the operation belongs to the current block, if

```

choose  $Op \subseteq \{op \mid kind(op) \in \{add, AND, read, write, jump, branch\}\}$  and
   $belongs\_to(op) = current\_block$  and  $value(op) = undef$  and
   $value(Pred_1(op)) \neq undef$  and  $(kind(op) \in \{add, AND, read, write, branch\}$ 
   $\Rightarrow value(Pred_2(op)) \neq undef)$  and
   $(kind(op) \in \{write, branch\} \Rightarrow value(Pred_3(op)) \neq undef)$  in  $\mathcal{P}(\text{Operations})$ 

  if  $op \in Op$  and  $kind(op) = add$  and  $\neg init$  then
     $value(op) := value(Pred_1(op)) + value(Pred_2(op))$  fi
  if  $op \in Op$  and  $kind(op) = AND$  and  $\neg init$  then
     $value(op) := value(Pred_1(op)) \wedge value(Pred_2(op))$  fi
  if  $op \in Op$  and  $kind(op) = read$  and  $\neg init$  then
     $value(op) := value(Pred_1(op))(value(Pred_2(op)))$  fi
  if  $op \in Op$  and  $kind(op) = write$  and  $\neg init$  then
     $value(op) := value(Pred_1(op))[value(Pred_1(op))(value(Pred_2(op)))] :=$ 
     $value(Pred_3(op))$  fi
  if  $kind(op) = jump$  and  $\neg init$  then
     $next\_block := Pred_1(Op)$ ;  $value(op) := def$  fi
  if  $kind(op) = branch$  and  $\neg init$ 
    then if  $value(Pred_1(op))$  then  $next\_block := Pred_2(op)$ 
    else  $next\_block := Pred_3(op)$  fi;  $value(op) := def$  fi
endchoose

```

Fig. 6. Transition Rule for Arithmetic, Boolean, Memory, Jump, Branch Operations

its predecessor operations are evaluated, if the ϕ nodes of the current block are evaluated ($\neg init$), and if the operation itself has not yet been evaluated, then the operation can be evaluated. Memory access operations are specified according to the functional store approach [Ste95]. Thereby, memory is itself treated as a global variable. Each write operation modifies the store and, hence, the value of this global variable. In SSA representations, modifications of variable values lead to duplications of that variable, i.e. to different copies, cf. also section 2. In case of the memory write operation, the result is a “new” memory which is identical with the old one except for the point which has been written by the write operation. In figure 6, the transitions for read and write operations are given. The read operation has two predecessor operations. The first gives the memory, the second the address to be read. The write operation has an additional predecessor, the value to be written to the address indicated by the second predecessor. Result of the read operation is the value stored in memory at the indicated address, result of the write operation is the updated memory.

```

if  $\{op \mid belongs\_to(op) = current\_block \text{ and } value(op) = undef\} = \emptyset$  then
   $pred\_block := current\_block$ ;  $current\_block := next\_block$ ;  $init := true$  fi

```

Fig. 7. Transition Rule for Block Transition

Jump and branch operations can also be evaluated if they belong to the current block and if the evaluation of the ϕ nodes has been completed ($\neg init$), even if there are other unevaluated operations (arithmetic, Boolean, memory read/write) in the basic block. The corresponding transition rules are also stated in figure 6. They set the value of the dynamic constant *next_block*. The transition to the succeeding basic block itself takes place by executing another transition rule, namely the one specified in figure 7. Its condition states that all operations must have been evaluated before transition to the next block can be done.

In summary, semantics of SSA representations can be described directly with ASMs. The imperative, state-based part is captured by transition rules which transfer control flow from one basic block to its successor block and by the distinction between the *init* phase which evaluates the ϕ nodes of the current block and the succeeding $\neg init$ phase which computes the remaining operations. This evaluation of the remaining operations is purely data-driven. It is captured very elegantly by non-deterministic transition rules such that at a given point during computation, more than one possible set of updates might be applicable.

5 Proof of Correctness for Code Generation in Compilers

Code generation in compilers transforms the intermediate representation into a sequence of machine instructions. When using SSA as intermediate form, one has to cope with imperative control flow between basic blocks as well as with the purely data-driven evaluation of operations within basic blocks. In contrast, the machine language is purely imperative because one instruction is executed after the other. To ensure correctness of machine code generation, one needs to prove that the semantics of the SSA representation is the same as the semantics of the generated machine code. For this proof, we need a formal semantics of the target machine language which we specify with ASMs in subsection 5.1. Typically, code generation is divided into code selection, instruction scheduling, and register allocation. To be able to concentrate on the essentials of the correctness proof, we consider a relatively simple code generation algorithm in subsection 5.2, prove its correctness in subsection 5.3, and discuss possible extensions in 5.4.

5.1 ASM Semantics for the Machine Language

We consider the machine language with the instructions summarized in table 1. Each instruction i may have a label l : " $l : i$ ". For simplicity, we assume that the machine can use arbitrarily many registers. The semantics of this machine language is formally specified with the ASM in figure 8. The static functions *NI* (next instruction), *LI* (labelled instruction), *TI* (true instruction), and *FI* (false instruction) are used to specify the order of instructions in the machine program. In the sequential case, one instruction is executed after the other. The function *NI* maps each instruction to its successor instruction. In case of JMP operations, the succeeding instruction is the one at the designated label. The function *LI* maps each JMP operation to this successor instruction. In

case of a BRN operation, we distinguish between the label in the positive and negative case, specified with the functions *TI* and *FI*. *TI* (*FI*) maps the current instruction to the instruction at label *l1* (*l2*). The dynamic functions *reg_table* and *Memory* map registers and memory addresses to their current content which is initially *undef*. The transition rules in figure 8 specify the dynamic semantics of the machine language. Thereby, the dynamic constant *curr_instr* serves as a pointer to the current instruction which is to be executed. In the initial states, it is initialized with the first instruction of the machine program.

Operation	Informal Semantics
ADD R1, R2, R3	adds register contents of R1, R2 and writes result in R3
AND R1, R2, R3	computes conjunction of contents of R1, R2 and writes result in R3
READ R1,R2	writes memory content at address stored in R1 into R2
WRITE R1, R2	writes value of R2 at address stored in R1
CP R1, R2	copies value of R1 into R2
JMP l	jumps to instruction with label l
BRN R1, l1, l2	branches to instruction with label l1 if content of R1 \neq 0, otherwise to instruction with label l2
NOP	does nothing

Table 1. Machine Language

```

if kind(curr_instr) = ADD then
  reg_table(R3) := reg_table(R1) + reg_table(R2); curr_instr := curr_instr.NI fi
if kind(curr_instr) = AND then
  reg_table(R3) := reg_table(R1)  $\wedge$  reg_table(R2); curr_instr := curr_instr.NI fi
if kind(curr_instr) = READ then
  reg_table(R2) := Memory(reg_table(R1)); curr_instr := curr_instr.NI fi
if kind(curr_instr) = WRITE then
  Memory(reg_table(R1)) := reg_table(R2); curr_instr := curr_instr.NI fi
if kind(curr_instr) = CP then
  reg_table(R2) := reg_table(R1); curr_instr := curr_instr.NI fi
if kind(curr_instr) = JMP then curr_instr := curr_instr.LI fi
if kind(curr_instr) = BRN then
  if reg_value(R1)  $\neq$  0
  then curr_instr := curr_instr.TI
  else curr_instr := curr_instr.FI
if kind(curr_instr) = NOP then curr_instr := curr_instr.NI fi

```

Fig. 8. ASM Semantics for Machine Language

5.2 Basic Code Generation Algorithm

Given an SSA program, machine code is generated in four phases. First, registers are assigned to the results of arithmetic, Boolean, ϕ , and memory operations. Then ϕ nodes are eliminated. Afterwards, machine code is generated separately for each basic block. Finally, global code generation unites the generated parts.

Assignment of Registers and Replacement of Operations: First, a fresh result register is assigned to each arithmetic, Boolean, read, and ϕ operation in the SSA form. Remember that we assume infinitely many registers. Then, these operations in the SSA form are replaced by machine instructions in the obvious way, e.g. *add* is replaced by ADD. The result of this transformation is a mix between the original SSA form and the machine language. The program is still in its SSA structure but its operations are already replaced by machine instructions.

Elimination of ϕ nodes: Then, the ϕ operations are eliminated in the standard way: If a block b contains the ϕ operation $x = \phi(x_1, \dots, x_n)$, then in each of the n predecessor blocks of b , x_i is copied into the same fresh register R : $CP(x_i, R)$, $1 \leq i \leq n$. Note that the x_i denote registers because in the first phase, we have assigned registers to the results of operations. In total, there are n such copy operations, and $CP(x_i, R)$ is placed in the i th predecessor block of b . In block b , the ϕ node is replaced by the operation $CP(R, R')$ for a fresh register R' . For a given basic block b , the result of this phase is denoted by E_b .

Code Generation for Basic Blocks: Afterwards, the arithmetic, Boolean, and memory operations within one basic block are arranged in a linear order which respects the data flow dependencies between them. Since the SSA form specifies only a partial order on the instructions, there are several valid linearizations. Each topological order of the acyclic SSA data flow graph is valid.

Problems arise if the SSA form contains memory write operations whose order is not enforced by the data flow dependencies. For the purpose of this paper, we assume that the write operations are already in a linear order in the SSA form. This assumption is natural because it can easily be met when transforming source programs into SSA form. The source program contains the write operations in a linear order so that the result memory (in the sense of functional stores) of a write operation is the input to the succeeding write operation. The read operations use some results of the write operations. These data flow dependencies determine the partial order between the read and write operations. Each topologic order of them is valid.

Concludingly, basic blocks are transformed into machine code by these steps:

1. First, each basic block b gets a unique label l_b .
2. Then, the arithmetic, Boolean, memory, and register assignment operations in a basic block b are sorted topologically into a linear sequence S_b of machine instructions.
3. Afterwards, the jump or branch operation contained in basic block b is replaced with a JMP or BRN machine instruction added to the end of S_b .

Thereby the labels LI , TI , and FI are replaced with the labels $l_{b'}$ assigned to the corresponding basic blocks b' which are denoted with LI , TI , and FI .

4. Finally, the instruction “ $l_b : \text{NOP}$ ” is inserted at the beginning of the sequence S_b of machine instructions of basic block b .

The complete machine program M_p is the concatenation of all sequences S_b for all basic blocks b in the SSA representation p such that S_{start_block} comes first.

5.3 Correctness of Basic Code Generation Algorithm

To prove the correctness of the code generation algorithm described in subsection 5.2, we distinguish between local and global correctness. Local correctness of code generation means informally that the results computed in a basic block b are the same, no matter if b is evaluated according to the SSA semantics or if b is first transformed into machine code S_b and then evaluated according to the machine language semantics. Global correctness means that local correctness holds for all basic blocks evaluated during execution. In the remainder of this subsection, we formalize these ideas.

The first step in the code generation algorithm assigns registers to the results of operations in the SSA form. Since we assume infinitely many registers, this assignment is described by an injective function $reg_assign : Operations \rightarrow Registers$ with the property: $op_1 \neq op_2 \Rightarrow reg_assign(op_1) \neq reg_assign(op_2)$. Furthermore, the SSA operations are replaced by corresponding machine instructions, described by the injective function $op_assign : \{add, AND, read, write, jump, branch\} \rightarrow \{ADD, AND, READ, WRITE, JMP, BRN\}$.

Definition 1 (Local Correctness). Let b be a basic block of an SSA representation, and let S_b be the machine code generated from it. b and S_b are semantically equivalent if, given that $value(op) = reg_table(reg_assign(op_assign(op)))$ for all $op \in \bigcup_{b' \in Preds(b)} b'$, it follows that for all $op \in b$, after evaluation of b and E_b , $value(op) = reg_table(reg_assign(op_assign(op)))$ holds. \diamond

Definition 2 (Global Correctness). Let p be an SSA representation with starting block $start_block$ and with $start_pred$ as predecessor block of $start_block$. Let M_p be the machine program generated from p . p and M_p are semantically equivalent if the following conditions hold:

- $value(op) = reg_table(reg_assign(op_assign(op)))$ holds for all op at the beginning of execution of p and M_p .
- Execution of p starts with $start_block$, execution of M_p starts with S_{start_block} .
- If the successor of a basic block b is b' , then the successor of S_b is $S_{b'}$. \diamond

To prove code generation locally and globally correct, we need a formal semantics for the mixed representation still exhibiting the SSA structure but containing already machine instructions. This is achieved with these modifications of the original ASM semantics for SSA:

Operations: The SSA operations *add*, *AND*, *read*, *write*, *jump*, *branch* are replaced by *op_assign*(*op*). E.g. *add* is replaced by *ADD*.

The dynamic function *value*: Throughout the ASM specification, the function *value* is replaced by *reg_table* \circ *reg_assign*.

Evaluation of ϕ nodes: The updates of the values of the ϕ nodes are replaced by the updates caused by the evaluation of the replacing copy operations. In general, a basic block might contain two kinds of copy operations, the ones in the beginning which replace the ϕ operations directly, and the ones at the end which place the input values for ϕ operations in succeeding blocks in the designated registers. When entering a basic block (i.e. *init* = *true*), only the first kind of copy operations is to be evaluated. Hence, we modify the transition rule of figure 5 as follows:

```

if init and current_block  $\neq$  void then
  forall op  $\in$  {op' | belongs_to(op') = current_block  $\wedge$  kind(op') = CP
     $\wedge$  kind(Pred1(op)) = CP}
  do in parallel value(op) = select(Preds(op), Pred(op)) od;
  forall op  $\in$  {op' | belongs_to(op') = current_block  $\wedge$  (kind(op')  $\neq$  CP  $\vee$ 
    kind(op') = CP  $\wedge$  kind(Pred1(op))  $\neq$  CP)}
  do in parallel value(op) = undef od;

```

Lemma 3 (ϕ Node Elimination). *Let b be an SSA basic block, E_b the corresponding transformed basic block in the mixed SSA/machine representation, and $value(op) = reg_table(reg_assign(op_assign(op)))$ for all $op \in \bigcup_{b' \in Preds(b)} b'$. Then $value(op) = reg_table(reg_assign(op_assign(op)))$ holds for all $op \in b$ after evaluation of b and E_b . \diamond*

Proof. The proof is by induction. Therefore the operations in b and E_b are partitioned into classes: The first class $C_{1,b}$ of b contains the ϕ operations of b , the first class C_{1,E_b} of E_b the corresponding CP operations. Given the first $1, \dots, i$ classes, the $i + 1$ st is defined as follows: It contains all those operations which can be evaluated if the values of the operations in the classes $1, \dots, i$ are known. Because a basic block in SSA form is an acyclic graph, there exists a smallest n_b such that all operations in b are uniquely partitioned into n_b classes. b and E_b are structurally nearly isomorphic: Each operation in b corresponds directly with an operation in E_b . Additionally, E_b has some copy operations at the end which place input values for ϕ operations in succeeding blocks in a special register. These copy operations at the end are placed in the class C_{n+1,E_b} .

Base Case: Let op be a ϕ operation in $C_{1,b}$. Assume that $op_assign(op) = CP\ R', R$. By assumption of lemma 3, the predecessors of op have the same values as the predecessors of $op_assign(op)$. The predecessor operations of $op_assign(op)$ in predecessor block $E_{b'}$ are operations $CP\ R_{b'}, R'$ which all copy a value in the same register R' . The content of R' is assigned to register R by $op_assign(op)$. This is the same value as one would get when executing the ϕ operation op in the original SSA form. Since all ϕ nodes and their substituting register copy operations are executed simultaneously, there is no interference between them.

Since all their input values exist, we conclude for all $op \in C_{1,b}$, $value(op) = reg_table(reg_assign(op_assign(op)))$ holds after execution of the corresponding transition rule in the original SSA and the modified SSA semantics.

Induction Case: For each class $C_{i,b}$ and its correspondent C_{i,E_b} , $2 \leq i \leq n$, it follows directly that $value(op) = reg_table(reg_assign(op_assign(op)))$ holds for all $op \in C_{i,b}$ after execution of the corresponding transition rules: There is always only one transition rule for each operation, and the value of the operation depends only on its input values which have been determined uniquely in the classes of operations evaluated before. Hence, we conclude for all $op \in C_{i,b}$ that $value(op) = reg_table(reg_assign(op_assign(op)))$ holds after execution of the corresponding transition rule in the original SSA and the modified SSA semantics by using the induction assumption that for all $op \in \bigcup_{1 \leq j \leq i-1} C_{j,b}$, $value(op) = reg_table(reg_assign(op_assign(op)))$ which completes the proof.

Note that for the operations in C_{n+1,E_b} , the input values are copied to the output values and serve as inputs of the ϕ operations in the succeeding block. Their purpose has been discussed already in the base case. ■

Lemma 4 (Local Correctness). *Let b be an SSA basic block and S_b the corresponding machine code. Then b and S_b are semantically equivalent.* ◇

Proof. Lemma 4 follows directly from lemma 3: The instructions in S_b are the same as the instructions in E_b . The linear order on the instructions in S_b contains the partial order on the instructions in E_b , i.e., an instruction is only executed in the schedule of S_b if its operand values have been computed before. ■

Theorem 5 (Global Code Generation). *Let p be an SSA representation with starting block $start_block$ and with $start_pred$ as predecessor block of $start_block$. Let M_p be the machine program generated from p . Then p and M_p are semantically equivalent.* ◇

Proof. We need to show that the three conditions of definition 2 are fulfilled. The first holds trivially because in the SSA semantics as well as in the machine semantics, all results of operation and values of registers, resp., are initialized with *undef*, except for the values of operations and registers in $start_pred$ and S_{start_pred} which are initialized with the input values of the program. The second condition follows in case of the SSA program directly from the ASM semantics for SSA because $current_block = start_block$ holds in the initial states. In case of the machine semantics, it follows from the fact that M_p starts with S_{start_block} and that execution starts with the first instruction.

A simple induction argument shows that the third condition holds for all basic blocks b and S_b executed during the run of p and M_p , resp.

Base Case: After execution of $start_block$ and S_{start_block} , $value(op) = reg_table(reg_assign(op_assign(op)))$ holds for all $op \in start_block$ (because of lemma 4), and hence for all op in p because execution of $start_block$ does not modify operations $op \notin start_block$. From the SSA and machine semantics, it follows directly that if the succeeding block in the execution of p is b' , then the succeeding block

in the execution of M_p is S_b . This holds because of the transition rule in figure 7, the algorithm for generating code for connecting basic blocks with JMP/BRN instructions using the labels LI , TI and FI , and the fact that the results of the *branch* and BRN as well as of the *jump* and JMP operations denote corresponding basic blocks in p and M_p .

Induction Case: Repeat the reasoning in the base case, thereby replacing *start_block* with an arbitrary current block b and S_{start_block} with S_b . ■

5.4 Discussion of Correctness Proof

The above correctness proof relates the SSA semantics which is partly non-deterministic with the strictly deterministic semantics of the machine language. Since the SSA semantics puts only a partial order on the operations in a given basic block, several valid linear orders are possible. Our proof shows that each of them is correct if one cares only about block-wise execution and is not interested in the intermediate states reached whenever only a subset of the operations in a basic block has been executed. In this sense, the machine language and the code generation algorithm that we have chosen in this paper, even though being simple, capture the essential problems when proving code generation correct. More advanced code selection algorithms might condense several succeeding operations in basic blocks into single complex machine instructions. This can easily be integrated into our correctness proof because the connection between the overall result of the corresponding subgraph in the SSA block and the result of the complex machine instruction can easily be established. Also code generation for VLIW (very long instruction word) processors can be integrated. In VLIW instructions, several data-independent instructions are executed simultaneously. Since SSA representations explicitly show the data dependencies, candidates for VLIW instructions can easily be identified. (SSA operations are data-independent if there is no directed path between them.) Hence, our correctness proof demonstrates the suitability of the stated ASM semantics for SSA representations and might serve as basis for further correctness proofs of more sophisticated optimizing code generation algorithms. Note that in such optimizing code generation algorithms, it might be necessary to undermine the concept of basic blocks by moving instructions from one basic block to another. Nevertheless, basic blocks are an inherent concept of SSA form and as such, they are part of our formal ASM semantics.

6 Related Work

ASMs have been used for the formal specification of many programming languages, cf. the detailed discussion in section 3. In [ZD03] a specification for the non-deterministic evaluation of expressions based on ASMs has been given which is similar to our specification for the non-deterministic data flow within basic blocks. Moreover, ASMs have been used successfully in proving the correctness of compilations, e.g. the correctness of compiling Prolog to the WAM

[BR94], the correctness of translating Occam to transputer code [BD96] and in the Verifix project which deals with the construction of provably correct compilers [ZG97, DV01, DvHVG02, GZ99]. While all these compilations are refining transformations, no systematic method is known for proving the correctness of optimizing compilers. In this paper, we have given a necessary prerequisite for such proofs, namely a formal semantics for SSA representations as well as a relatively simple correctness proof for machine code generation extendable to capture also non-refining optimizing transformations.

7 Conclusions

In this paper, we have stated a formal semantics for SSA representations in a simple and elegant way based on ASMs. We have described the state-based imperative part of SSA computations by transition rules which transfer control flow from the current to the succeeding basic block. Furthermore we have specified the purely data flow driven computation within basic blocks by transition rules which are non-deterministic in the sense that during the evaluation of a basic block more than one set of updates might be applicable. Each specification should demonstrate its usefulness in order to not become an end in itself. We have provided such a demonstration of usefulness by showing that our specification can serve as basis in proofs of correctness for machine code generation.

In future work, we want to prove the correctness of more elaborate code generation algorithms. In particular, we want to extend the machine language to include very long instruction words (VLIW), predicated instructions and speculative execution. This also implies that the code generation algorithm be extended to generate machine code optimized for such instruction sets. Moreover, we want to drop the assumption that there are infinitely many registers by considering optimizing register allocation algorithms as well. Furthermore, we also want to prove the correctness of data flow analyses and corresponding machine independent optimizations of SSA representations. These are optimizations which transform a given SSA form into a semantically equivalent SSA form, e.g. by eliminating dead code or common subexpressions. For all these correctness proofs, the formal SSA semantics and the correctness proof stated in this paper are supposed to serve as basis, as already discussed in subsection 5.4. For many of these optimizations, it is necessary to move instructions between basic blocks. Such transformations will need more sophisticated proof techniques since then, it is harder to identify corresponding states in the original and optimized program.

Acknowledgments: The author would like to thank Wolf Zimmermann and Jan Olaf Blech for many fruitful discussions on the topics of this paper. Also thanks to the anonymous reviewers for their helpful comments.

References

- [BD96] Egon Börger and Igor Durdanovic. Correctness of compiling Occam to Transputer Code. *Computer Journal*, 39(1):52–92, 1996.

- [BR94] Egon Börger and Dean Rosenzweig. The WAM - definition and compiler correctness. In L.C. Beierle and L. Pluemer, editors, *Logic Programming: Formal Methods and Practical Applications*. North-Holland Series in Computer Science and Artificial Intelligence, 1994.
- [CFR⁺91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [DV01] Axel Dold and Vincent Vialard. A Mechanically Verified Compiling Specification for a Lisp Compiler. In *Proceedings of the 21st Conference on Software Technology and Theoretical Computer Science (FSTTCS 2001)*, 2001. Springer Verlag, Lecture Notes in Computer Science, Vol. 2245.
- [DvHVG02] A. Dold, F. v. Henke, V. Vialard, and W. Goerigk. A Mechanically Verified Compiling Specification for a Realistic Compiler. Ulmer Informatik-Berichte 02-03, Universität Ulm, Fakultät für Informatik, 2002.
- [EGGP00] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of Design Languages: A compilation approach using Abstract State Machines. In *Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines*. Local Proceedings, TIK-report 87, Swiss Federal Institute of Technology (ETH) Zurich, 2000.
- [GH93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In *Selected papers from CSL'92 (Computer Science Logic)*, pages 274–308. Springer Verlag, LNCS Vol. 702, 1993.
- [Gur95] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.
- [GZ99] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In *Correct System Design*. Springer-Verlag, LNCS Vol. 1710, 1999.
- [Muc97] Steven S. Muchnick. *Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.
- [New01] Heise Newsticker. *Rotstich durch Fehler in Intels C++ Compiler*. <http://www.heise.de/newsticker/data/hes-11.11.01-000/>, 2001.
- [SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer Verlag, 2001.
- [Ste95] B. Steensgaard. Sparse Functional Stores for Imperative Programs. In *The First ACM SIGPLAN Workshop on Intermediate Representations*, 1995.
- [Wal95] Charles Wallace. The Semantics of the C++ Programming Language. In *Specification and Validation Methods*. Oxford University Press, 1995.
- [ZD03] W. Zimmermann and A. Dold. A Framework for Modeling the Semantics of Expression Evaluation with Abstract State Machines. In *Abstract State Machines - Advances in Theory and Applications, Proc. of the 10th International Workshop, ASM 2003*, 2003. Springer Verlag, LNCS Vol. 2589.
- [ZG97] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Backends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.