Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Electronic Notes in Theoretical Computer Science 141 (2005) 33–51

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL

Jan Olaf Blech Sabine Glesner Johannes Leitner
Steffen Mülling

Institute for Program Structures and Data Organization
University of Karlsruhe, 76128 Karlsruhe, Germany

Abstract

Correctness of compilers is a vital precondition for the correctness of the software translated by them. In this paper, we present two approaches for the formalization of static single assignment (SSA) form together with two corresponding formal proofs in the Isabelle/HOL system, each showing the correctness of code generation. Our comparison between the two proofs shows that it is very important to find adequate formalizations in formal proofs since they can simplify the verification task considerably. Our formal correctness proofs do not only verify the correctness of a certain class of code generation algorithms but also give us sufficient, easily checkable correctness criteria characterizing correct compilation results obtained from implementations (compilers) of these algorithms. These correctness criteria can be used in a compiler result checker.

Keywords: formal compiler correctness, SSA representation, optimizing code generation, compiler result checker, Isabelle/HOL.

1 Introduction

Compiler correctness is a necessary prerequisite to ensure software correctness and reliability as most modern software is written in higher programming languages and needs to be translated into native machine code. In this paper, we address the problem of verifying compiler correctness formally within the theorem prover Isabelle/HOL [20]. Starting from intermediate representations in static single assignment (SSA) form, we consider optimizing machine code generation based on bottom-up rewrite systems. To prove the correctness of

doi:10.1016/j.entcs.2005.02.042

such program transformations, a formal semantics of the involved programming languages, i.e. of the SSA intermediate representation form as well as of the target processor language, is necessary. Furthermore, a formal proof[1] is required that shows that the transformations preserve the semantics of the compiled programs. Such proofs only deal with transformation algorithms themselves but not with a given compiler implementing them. To bridge this gap, we require the formal proofs to deliver sufficient, easily checkable correctness conditions that classify if a compilation result is correct.

Our solution is based on the observation that SSA programs specify imperative, i.e. state-based computations. In a previous work [11], we have shown that SSA semantics can be captured elegantly and adequately with abstract state machines [12]. In this paper, we show that this semantics can be expressed within the theorem prover Isabelle/HOL and that correctness of code generation can be formally shown based on it, also within Isabelle/HOL. The formalization of SSA in Isabelle/HOL offers many degrees of freedom, in particular the formalization of the data-flow driven computations within SSA basic blocks. In a previous work [2] (which we summarize here), we have represented basic blocks by *term graphs* [5]. Term graphs represent acyclic graphs by duplicating common subexpressions. To keep track of duplicates, we have assigned a unique identification number to each node in the original graph and kept these numbers when duplicating common subexpressions in order to be able to identify identical subexpressions in the term graphs. Based on this formalization, we have defined a formal semantics for SSA basic blocks by stating a function that evaluates term graphs. In this paper, we present a new approach. Basic blocks in our second approach are directly represented as partial orders rather than as term graphs. We show that this second formalization cannot only be handled easier in theorem provers but also that proofs for this second formalization can be reused in other areas of software verification as well since it is by far more general.

This paper is organized as follows: In Section 2, we introduce SSA form. Then we summarize our previous work on the verification of code generation based on term graphs: Therefore, in Section 3, we explain the formal semantics of SSA within Isabelle/HOL and in Section 4 the correctness proof for a relatively simple code generation algorithm. In Section 5, we present our novel proof approach based on partial oders. In Section 6, we compare our two formalizations and point out why the second is much better. In Section 7 we show that the verified correctness criterion can be integrated into the compiler checker approach. Related work is discussed in Section 8. Finally, in

[1] We denote proofs in theorem provers with the term *formal proofs*, in contrast to "paper and pencil-proofs".

Section 9, we conclude and discuss future work.

2 Static Single Assignment Intermediate Languages

Static single assignment (SSA) form has become the preferred intermediate representation for handling all kinds of program analyses and optimizing transformations prior to code generation [6]. 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.

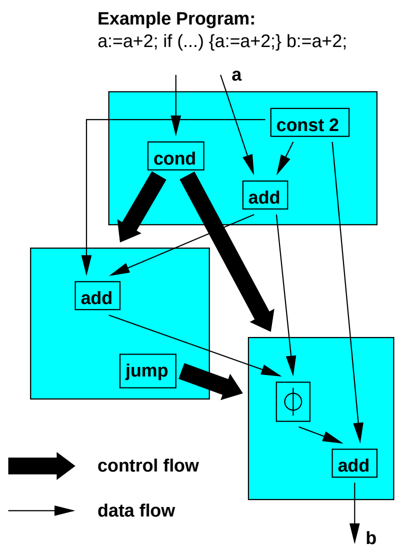
By definition SSA-form requires that a program and in particular each basic block is represented as a directed graph of elementary operations (memory read/write, jump/branch, arithmetic operations on data) such that each "variable" is assigned exactly once in the program text. Only references to such 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 and 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 block, ϕ *nodes*, $x = \phi(x_1, \ldots, x_n)$, represent the unique value assigned to variable x. This value is a selection among the values $x_1, \ldots, x_n$ where x_i represents the value of x defined on the control path through the i-th predecessor of the basic block. The number of predecessors of this block is n. Programs can easily be transformed into SSA form [16], e.g. by a tree walk through the attributed syntax tree. The standard transformation subscripts each variable. At join points, ϕ nodes sort out multiple assignments to a variable corresponding to different control flows of the program.

As example, the figure to the left shows the SSA form 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. 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 since in SSA form, 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 block, control flow



is transferred to the uniquely defined subsequent block. Hence, the current state is characterized by the current basic block and by the outcomes of the operations in the previously executed basic blocks.

3 A Formal Semantics of SSA Based on Term Graphs

In this section we describe our previous work [2] concerning the formalization of SSA semantics within Isabelle/HOL based on term graphs. In Subsection 3.1, we present our specification of basic blocks and in Subsection 3.2 our formalization of the global control and data flow.

Isabelle is a generic interactive theorem prover that can be instantiated with different logics. Its instantiation Isabelle/HOL with higher-order logic (which allows for the quantification over functions and predicates) has adequate expressive power for the specification of and reasoning about programming languages and also provides a series of predefined helpful theories. In particular, Isabelle/HOL offers possibilities to define data types inductively as well as to define functions by primitive or general terminating recursion.

3.1 Formal Semantics of Basic Blocks

Basic blocks in SSA intermediate representations can be regarded as directed acyclic graphs (DAGs) such that the nodes represent operations (e.g. arithmetic operators, constants, or ϕ nodes) and the edges represent the data flow in between. Evaluation of basic blocks takes place in two steps: First, the ϕ nodes are evaluated simultaneously. Then, the results of the remaining

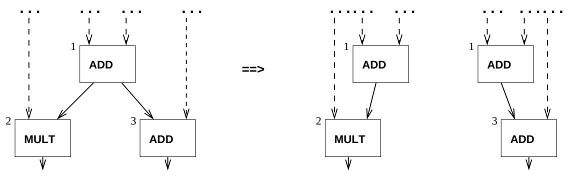


Fig. 1. Transforming SSA DAGs into SSA Trees

```
consts
  eval_tree :: "SSATree ⇒ SSATree"
primrec
  "eval_tree (CONST val ident) = (CONST val ident)"
               . . . . .
  "eval_tree (NODE operator tree1 tree2 val ident)  =
    (NODE operator (eval_tree tree1) (eval_tree tree2)
    (operator (get_ssatree_val (eval_tree tree1)) (get_ssatree_val (eval_tree tree2)))
    ident)"        . . .
```

Fig. 2. Evaluation of SSA Expressions

operations are determined. We specify the first step, evaluation of ϕ nodes, together with the global control flow, cf. subsection 3.2. Therefore we can treat ϕ nodes within a given basic block as constants. Hence, constants and ϕ nodes (within a given basic block) are nodes with only outgoing edges.

DAGs representing SSA basic blocks contain common subexpressions only once. In our formalization based on term graphs, we have represented such a DAG by transforming it into an equivalent set of trees by duplicating shared subterms, cf. Figure 1. To enable identification of equivalent subtrees, we assign a unique number to each operation in the original DAG and duplicate this identification number whenever duplicating a shared subexpression. We can transform such a set of trees into a single tree by adding a root node. In Isabelle/HOL, these trees are formalized in the following manner:

```
datatype SSATree = CONST value identifier | PHI phiargs value identifier |
    NODE operator SSATree SSATree value identifier | ···
```

Nodes represent constants, ϕ-nodes with argument lists and arithmetic operations. We have also formalized memory operations (hence the little dots), but since these extra features are not in the focus of this paper we refer to [2,4] for a detailed presentation. SSA basic blocks are evaluated with the evaluation function *eval_tree* defined inductively on SSA trees, cf. Figure 2.

Remark: Because *CONST* and *PHI* nodes behave the same when processed by *eval_tree* within a fixed basic block, we treat them uniformly as *LEAF* in the proof in section 4.

3.2 Formal Semantics for the Global Control and Data Flow

An SSA program is formalized as a list of basic blocks whereby each basic block carries four pieces of information which integrates it into the global control and data flow:

datatype *BASICBLOCK* =
  NEW identifier "identifier $\times$ *nat" "identifier* $\times$ *nat" "SSATree list"*

These four entries describe the following information:

1. *identifier*	the value number determining the successor basic block
2. *identifier* $\times$ *nat*	successor target 1 and its rank
3. *identifier* $\times$ *nat*	successor target 2 and its rank
4. *SSATree list*	list of SSATrees with the operations of the basic block

In our formalization, a basic block b can have two different successors b' (target 1 and target 2) specified by the third and fourth field of type *identifier* $\times$ *nat*. *identifier* is the number characterizing the successor block. *nat* specifies its rank which defines the selection of the arguments in the ϕ nodes in b': If the value of rank is i, then the ith argument in the argument list of each ϕ node in b' is chosen. (Remember that ϕ nodes have exactly as many operands as the basic block has predecessor blocks.)

Execution of SSA programs is state-based. Each single state transition corresponds to the execution of a single basic block. We define the current state by the values of the operations executed in previous basic blocks and by the currently executed basic block. Therefore we specify a state as:

- a table of values	formalized as function (*identifier* $\Rightarrow$ *value*) indexed by value numbers
- current basic block and its rank	

The state transition function (*step* :: *"BASICBLOCK list* $\Rightarrow$ *state* $\Rightarrow$ *state"*) evaluates basic blocks by performing the following computations:

- it assigns each ϕ-node its value
- it evaluates the basic block (i.e. calculates and stores values in nodes)
- it collects all calculated values and updates the table of values
- it determines the successor basic block (from the corr. distinct value number)

We have specified the semantics of SSA intermediate languages via this state transition function, thereby covering all major aspects of SSA based languages. For a complete specification with all details, we refer to [4].

4 Verification of Code Generation with Term Graphs

In this section, we consider a relatively simple code generation algorithm and prove part of its correctness by showing that it preserves the observable behavior of translated basic blocks. Therefore, as core of the proof, we show that every topological sorting of a basic block is a correct code generation order. This is the most interesting part in the overall correctness proof for code generation as it transforms the tree or DAG structure, resp., into a linear code sequence. Furthermore, since we prove correctness of code generation for individual basic blocks, we can treat *PHI* nodes as constants and, hence, do not distinguish between *PHI* and *CONST* nodes but instead treat them uniformly as *LEAF* nodes.

4.1 Semantics of the Machine Language

Machine code is formalized as a list of CodeElements which operate on values stored in a *value table* which can be considered as an infinite set of registers holding the results of all hitherto computed value numbers. The value table is specified as a function (*identifier* ⇒ *value*) that maps identifiers to their current values. Since we concentrate on the correct translation of individual basic blocks, it is sufficient to work with this machine language:

datatype *CodeElement* = *L value identifier* | *N operator identifier identifier identifier*

The "*L value identifier*"-element has the following semantics: store *value* at value table cell specified by *identifier*. The "*N operator identifier identifier identifier*"-element means: get value stored at first *identifier*, get value stored at second *identifier*, apply *operator* on both values and store the result at the third *identifier*. The function that evaluates a machine code list updates the value table:

eval_codelist :: "*CodeList* ⇒ (*identifier* ⇒ *value*) ⇒ (*identifier* ⇒ *value*)"
and is primitive recursive over the code list and evaluates one instruction after the other.

4.2 Proof Prerequisites: Translation Function and Topsort Criterion

Prerequisites for our proof are twofold: First, we need to specify the translation between SSA form and the machine language. Secondly, we need to define the predicate *is_topsort* which describes the sequences of machine code that preserve the partial order on the operations determined by SSA basic blocks. Concerning the first need, the translation function, we have defined a function *ce_ify*[2] that maps an SSATree (node) to a code element (*SSATree* ⇒ *CodeElement*). Our formalization of topological sortings, formally defined by the predicate *is_topsort*, covers the following aspects:

[2] *ce_ify* stands for *CodeElementify*.

- Each element in the tree must have a corresponding element in the code list.
- Each element in the code list must have a corresponding element in the tree.
- If an element a in the tree is a successor of another element b, then the corresponding element $ce\_ify\ a$ must also be a successor of $ce\_ify\ b$ in the code list.
- Each element in the code list has a unique identifier.

A detailed description of the Isabelle/HOL specification defining these requirements can be found in [4]. As example, the first requirement is formalized in Isabelle/HOL by:

$$(\forall\ a.((is\_in\_tree\ a\ tree) \longrightarrow (\exists\ b.((is\_in\_cl\ b\ clist) \wedge (ce\_ify\ a = b))))).$$

The predicate is_in_cl ($CodeElement \Rightarrow CodeElement\ list \Rightarrow bool$) holds if $CodeElement$ is contained in $CodeElement\ list$. The predicate is_in_tree ($SSATree \Rightarrow SSATree \Rightarrow bool$) is defined analogously for the subtree relation.

4.3 The Main Theorem

We claim that if a code list is a topological sorting of an SSA tree, then each value calculated in the tree must also be calculated in the code list and stored under the same value number in the value table: [3]

theorem main theorem**:**

$$\begin{aligned}
&''(\forall\ clist.\ ((is\_topsort\ clist\ tree) \longrightarrow \\
&\quad (\forall\ t.(is\_in\_tree\ t\ (eval\_tree\ tree)) \longrightarrow \\
&\qquad (\exists\ ident\ val.(val = (eval\_codelist\ clist(\lambda\ a.(Eps(\lambda\ a.\ False)))\ ident) \wedge \\
&\qquad\ (val = get\_ssatree\_val\ t) \wedge (ident = get\_ssatree\_id\ t)))))''
\end{aligned}$$

Proof of main theorem: By induction over the SSATree *tree*:

Proof of Base Case: We show that if $is\_topsort\ clist\ (LEAF\ val\ ident)$ holds, then the result of $LEAFval\ ident$ is also computed by the machine program and is available under value number $ident$ after execution of $clist$, cf. [2] for more details.

Proof of Induction Step: Proving the induction-step is more difficult. We have the following induction assumptions:

- $\forall\ list'.is\_topsort\ list'\ kid1 \Longrightarrow$ every value calculated in $kid1$ is calculated in $list'$.
- $\forall\ list''.is\_topsort\ list''\ kid2 \Longrightarrow$ every value calculated in $kid2$ is calculated in $list''$.

and need to show that:

$\forall\ list.is\_topsort\ list\ (NODE\ fun\ kid1\ kid2\ val\ ident) \Longrightarrow$ every value calculated in $(NODE\ fun\ kid1\ kid2\ val\ ident)$ is also calculated in $list$.

[3] *Eps* denotes the Hilbert ϵ-operator defined in Isabelle/HOL which embodies the axiom of choice.

In our proof, we have skolemized the ∀-quantified variables *list′* and *list″* in the induction assumptions by instantiating them with *proj list kid1* and *proj list kid2*. The function (*proj* :: *"CodeElement list ⇒ SSATree ⇒ CodeElement list"*) maps all elements from the input CodeElement list having a corresponding element in the SSATree to the output code element list. In our proof we have defined the *proj* function via its properties. From these characteristics and from the induction hypotheses, we can derive that every value that gets calculated in *kid1* and *kid2* will be calculated in the CodeElement list *list*.

To complete the proof, for every subtree t of *tree*, we show that its values are calculated in the code list. We prove this by the following case distinction: t is subtree of *kid1*, or t is subtree of *kid2*, or t is the root node: *tree*.

The first two cases can be derived from the induction hypotheses and the characteristics of the *proj* function. For the third case, we show that for every topologically sorted list of a tree the last element corresponds to the root. Since every child node is correctly evaluated in the CodeElement list, we derive that the root node is also evaluated correctly. □

Summary of Achievements:

In total, our proof in Isabelle/HOL has required 885 lines of proof code with 45 lemmas and the main theorem. This is already a large proof, in particular if one takes into account that proofs in HOL need to be done interactively by hand. Moreover, at certain points during the proof, we noticed that proof steps were unnaturally complicated. These difficulties arose because we have different induction principles in the SSA trees and in the machine code lists which do not correspond directly with each other. For this reason, we have developed an alternative formulation of SSA basic blocks in which we represent them directly as partial orders. In this formalization, code generation is correct if the order in the generated code is contained in the original partial order of the basic blocks. While the proof based on term graphs, which we have presented in this section, captures the operational character of SSA basic blocks more intuitively, the formalization with partial orders allows for a correctness proof which is based on simpler mathematical concepts. We describe the proof based on partial orders in the following section.

5 A Formal Semantics for SSA Based on Partial Orders

In the previous section, we have presented our proof for the correctness of code generation from SSA form based on the formalization of SSA basic blocks as term graphs. While this formalization captures the operational character of

SSA blocks very well, it does not reflect their data flow driven nature equally clearly. In our proof, this deficiency has shown up in many surprisingly complicated proof details resulting from the fact that we have had different induction principles in the SSA blocks (induction on trees) and the generated sequences of machine code (induction on lists). However, data flow driven computations are characterized simply by the fact that operations can be performed as soon as their operands are available. In this view, SSA basic blocks turn out to be equivalent to partial orders. Input values and constant functions of a block make up the elements of the partial order which do not have predecessors. An operation taking results of other operations as input is an element that has predecessors in the partial order, namely all those operations whose outcome is directly or transitively necessary for its evaluation.

We have taken this intuition as basis for an alternative formalization of SSA basic blocks based on partial orders. We refer to [3] for our complete collection of definitions and proofs which we have formalized in Isabelle/HOL. In the rest of this section, we summarize the most interesting parts of it.

5.1 Definition and Basic Properties of Partial Orders

We formalize partial orders based on the notion of relational sets. A relational set *RelSet* is defined as a tuple consisting of a carrier set and of a set of tuples containing all elements of the carrier set that are in relation to each other:

types $'a\ RelSet\ =\ 'a\ set\ \times\ ('a \times' a)\ set$

A relation *RelSet* is sane if all elements occurring in the tuples of the relation are also contained in the carrier set of the relation (*fst* and *snd* denote the first and second entry in a tuple):

constdefs
$sane\ ::\ 'a\ RelSet\ \Rightarrow\ bool$
$sane\ RS\ \equiv\ \forall\ a\ b.\ (a,b)\ \in\ snd\ RS\ \longrightarrow\ a\ \in\ fst\ RS\ \wedge\ b\ \in\ fst\ RS$

In the following, basic properties (antireflexivity, being a strict partial order, being a strict finite partial order) of relational sets are formalized:

constdefs
$antirefl\ ::\ ('a\ \times' a\ )\ set\ \Rightarrow\ bool$
$antirefl\ r\ \equiv\ \forall\ x.\ (x,x) \notin r$
$spo\ ::\ 'a\ RelSet\ \Rightarrow\ bool$
$spo\ RS\ \equiv\ trans\ (\ snd\ RS\ )\ \wedge\ antirefl\ (\ snd\ RS\ )\ \wedge\ sane\ RS$
$sfpo\ ::\ 'a\ RelSet\ \Rightarrow\ bool$
$sfpo\ RS\ \equiv\ finite\ (\ fst\ RS\ )\ \wedge\ spo\ RS$

The minus operator $\ominus$ deletes an element from a relational set by removing it from its carrier set and by furthermore removing all tuples containing this element. With the **infixr** declaration we have defined $\ominus$ as an infix abbreviation for *sfpo_minus* with an appropriate priority:

constdefs
$sfpo\_minus :: {'a\ RelSet} \Rightarrow {'a} \Rightarrow {'a\ RelSet}$ (**infixr** $\ominus$ 65)
$RS \ominus x \equiv ((fst\ RS) - \{x\}, (snd\ RS) - \{(a,b) . (a,b) \in (snd\ RS) \wedge ((a = x) \vee (b = x))\})$

The function *sfpo_union* adds additional dependencies into a relational set (where ˆ+ denotes the transitive closure):

constdefs
$sfpo\_union :: [{'a\ RelSet}, ({'a} \times {'a})\ set] \Rightarrow {'a\ RelSet}$ (**infixr** $\cup$ 999)
$sfpo\_union\ rs\ r \equiv (fst\ rs, (snd\ rs \cup r)\hat{}+)$

Since we want to formalize SSA blocks by partial orders, we need the notion of the predecessors of an element (i.e. the input values of an SSA operation). We have done this with the following definitions and lemmata. The first definition specifies the predecessors of an element in the carrier set of a relation in the natural way:

constdefs
$predecessors :: [({'a} \times {'a})\ set, {'a}] \Rightarrow {'a}\ set$
$predecessors\ rel\ x \equiv \{y . (y,x) \in rel\}$

Besides some minor auxiliary lemmata (documented in [3]), we have verified the following theorems. They state properties about partial orders when elements are removed from them. The two theorems say that if one removes an element from a strict (finite) partial order, then it remains a strict (finite) partial order:

theorem *remove_one* : $spo\ RS \Longrightarrow spo\ (RS \ominus m)$
theorem *sfpo_remove_invariant* : $sfpo\ RS \Longrightarrow sfpo\ (RS \ominus m)$

5.2 Closure Operations and Reachability on Finite Partial Orders

In this subsection, we define and prove some important properties of finite partial orders. In particular, we introduce the notion of reachability in finite partial orders and prove some important properties of it. In Subsection 5.3, we use the results of this section directly to prove the correctness of code generation for SSA basic blocks.

With the notion of reachability, we classify which elements in a partial order can be reached from a given set of basic elements. An element is reach-

```
theorem sfpo_reachable_complete :
  assumes relation_is_sfpo : sfpo RS
  shows fst RS = reachable RS
proof −
  have s1 : ⋀ RS n. ⟦ sfpo RS ; card (fst RS) = n ⟧ ⟹ fst RS ⊆ reachable RS
  proof −
    fix n show ⋀ RS . ⟦ sfpo RS ; card (fst RS) = n ⟧ ⟹ fst RS ⊆ reachable RS
    proof ( induct n )
      case 0
      hence fst RS = {} by ( rule sfpo_zero_empty )
      thus ?case by auto
    next
      case ( Suc i )
      assume is_sfpo : sfpo RS and si_is_card : card (fst RS) = Suc i
      have ⋀ x . x ∈ fst RS ⟹ x ∈ reachable RS
      proof −
        fix x assume x ∈ fst RS
        hence card ( fst ( RS ⊖ x ) ) = i using is_sfpo and si_is_card
          by ( subgoal_tac finite ( fst RS ) , simp add : sfpo_minus_def remove_card ,
            simp add : sfpo_imp_finite )
        moreover have sfpo ( RS ⊖ x ) using is_sfpo
          by ( rule sfpo_remove_invariant )
        ultimately have induction_case : fst ( RS ⊖ x ) ⊆ reachable ( RS ⊖ x )
          by ( simp add : Suc )
        hence ∀ p. p ∈ predecessors (snd RS) x ⟶ p ∈ reachable RS
        proof auto
          fix p assume p_is_pred : p ∈ predecessors (snd RS) x
          hence p ∈ fst ( RS ⊖ x ) using is_sfpo by ( rule predecessor_in_remainder )
          hence p ∈ reachable ( RS ⊖ x ) using induction_case by auto
          moreover have x ∉ predecessors ( snd RS ) p using p_is_pred and is_sfpo
            by ( rule_tac y = p in predecessors_anti , auto )
          moreover have x ≠ p using p_is_pred and is_sfpo
            by ( rule_tac y = p in predecessors_notself , auto )
          ultimately show p ∈ reachable RS using is_sfpo
            by ( rule_tac x = p and z = x in invariant_if_smaller , auto )
        qed
        thus x ∈ reachable RS by ( rule reachable.step )
      qed
      thus ?case by auto
    qed
  qed
  hence fst RS ⊆ reachable RS using relation_is_sfpo by ( simp add : s1 )
  moreover have reachable RS ⊆ fst RS by
   ( simp add : reachable_has_only_nodes )
  ultimately show fst RS = reachable RS by simp
qed
```

Fig. 3. Proof for Theorem *sfpo_reachable_complete* in Isar Style Notation

able if all its predecessors are reachable. Applied to the context of SSA basic blocks, this means that an operation (which is represented by an element in a suitable partial order) can be evaluated if all its inputs (wich are predecessors in the corresponding partial order) have been computed already. Elements without predecessors are by default reachable; they correspond to constant operations (constant functions or already evaluated ϕ nodes in an SSA block).

The function *reachable* computes for a relation the set of elements that are reachable. These are all elements without any predecessors plus those elements that can be also transitively reached from them:

consts *reachable* :: *'a RelSet* $\Rightarrow$ *'a set*
inductive *reachable RS*
  intros
  step : $[\![\forall\, y\,.\, y \in predecessors\ (\ snd\ RS\ )\ x \ \rightarrow\ y \in reachable\ RS\ ;\ x \in fst\ RS\ ]\!]$
   $\Longrightarrow\ x \in reachable\ RS$

The following two lemmata are necessary to prove the theorem *sfpo_reach−able_complete* about finite partial orders which we discuss directly afterwards. The first lemma *reachable_has_only_nodes* states that all reachable elements are nodes, i.e. are contained in the carrier set of the relation. The second lemma *invariant_if_smaller* says that if one adds an element z to a relation which does not contain z already and if z does not become a predecessor of another reachable element x, then x stays reachable.

lemma *reachable_has_only_nodes* : $reachable\ RS \subseteq fst\ RS$
lemma *invariant_if_smaller* :
  $[\![\ x \in reachable\ (\ RS \ominus z\ )\ ;\ sfpo\ RS\ ;\ z \notin predecessors\ (\ snd\ RS\ )\ x\ ;\ z \neq x\ ]\!]$
  $\Longrightarrow\ x \in reachable\ RS$

With the formalization presented so far, we are ready to state and prove one of our main statements, namely the theorem *sfpo_reachable_complete*. It states that if a relation is a partial order, then the set of reachable elements equals the carrier set of the relation:

theorem *sfpo_reachable_complete* : $sfpo\ RS\ \Longrightarrow\ reachable\ RS\ =\ fst\ RS$

The proof of the above theorem is displayed in Figure 3. This proof is central in our formalization. Speaking in terms of data flow dependencies, this means that all operations can be computed because the operands of each of them will eventually be available during computation.

5.3 SSA Basic Blocks as Partial Orders

We define values in the SSA form as being of a generic value type *valueType*:

```
datatype 'a valueType = Undefined | V 'a        types nodeId = nat
```

Furthermore, we represent SSA basic blocks as annotated partial orders. Each SSA basic block is a structured entity that consists of a set of node identifiers, of a function that maps each node identifier to the list of the identifiers of its argument nodes (i.e. to its predecessor nodes), and of a function that returns for each node an associated operation:

```
record 'a ssa_graph =
  base :: nodeId set
  plist :: nodeId ⇒ nodeId list
  function :: nodeId ⇒ ( 'a valueType list ⇒ 'a valueType )
```

This definition corresponds to the relation RelSet we defined earlier. Now we are ready to define the evaluation of SSA graphs. This definition is formulated inductively. Isabelle allows for the inductive definitions of sets by arbitrary monotone functions. The two lemmata in the definition in Figure 4 state such monotonicity properties. The first lemma *evalssa_mono_helper* uses the predicate *list_all2* which is predefined in Isabelle/HOL as follows:

$$list\_all2\ P\ xs\ ys\ \equiv\ length\ xs\ =\ length\ ys\ \wedge\ (\ \forall\ (x,y) \in set\ (\ zip\ xs\ ys\ )\ .\ P\ x\ y\ ).$$

The first lemma says that if all pairs of elements contained in a certain set (in *set (zip l1 l2)*) are elements of A, then they are also elements of B, given that $A \subseteq B$; a rather trivial monotonicity property. The second lemma states a similar monotonicity fact. By using the key word **monos** followed by the names of the two lemmata, we give Isabelle the instruction to use these lemmata when proving the validity of the inductive definition.

The inductive set *evalssa* is defined in Figure 4 in dependence of its two parameters *ssaG* and *eval_order*. *ssaG* denotes an SSA graph and *eval_order* a set of pairs of nodes comprising additional evaluation dependencies. The inductive rule of the definition of *evalssa* states three preconditions. The first precondition,

$$list\_all2\ (\lambda x\ y.(x,y) \in (\ evalssa\ ssaG\ eval\_order\ ))\ (\ plist\ ssaG\ x\ )\ list\_of\_values$$

states that all predecessors of x must already be in *evalssa* by also having appropriate values (elements of the list *list_of_values*) associated with them. The second precondition,

$$\forall p.(p,x) \in eval\_order\ \longrightarrow\ (\ \exists v.\ (p,v) \in\ (\ evalssa\ ssaG\ eval\_order\ ))$$

requires that not only the dependencies contained in the SSA graph are respected but moreover also all dependencies expressed with the extra set *eval_order*. If there is a dependency $(p,x) \in eval\_order$, then p must already be contained in *evalssa*. And finally the third precondition,

consts $evalssa$ ::
  $[\ 'a\ ssa\_graph\ ,\ (\ nodeId \times nodeId\ )\ set\ ] \Rightarrow (\ nodeId \times\ 'a\ valueType\ )\ set$
lemma $evalssa\_mono\_helper\ :\ A \subseteq B \Rightarrow list\_all2\ (\lambda x\ y.\ (x,y) \in A)\ l1\ l2 \rightarrow list\_all2\ (\lambda x\ y.\ (x,y) \in B)\ l1\ l2$
lemma $evalssa\_mono\_helper2\ :\ A \subseteq B \Rightarrow (\lambda(x,y).(y,v) \in A)\ z \rightarrow (\lambda(x,y).(y,v) \in B)\ z$

inductive $evalssa\ ssaG\ eval\_order$
intros
  $step$:
  $[\![list\_all2\ (\lambda x\ y.(x,y) \in (\ evalssa\ ssaG\ eval\_order\ ))\ (\ plist\ ssaG\ x\ )$
    $list\_of\_values\ ; \forall p.(p,x) \in eval\_order \longrightarrow$
   $(\ \exists v.\ (p,v) \in\ (\ evalssa\ ssaG\ eval\_order\ ))\ ;\ x \in base\ ssaG\ ]\!] \Longrightarrow$
    $(\ x\ ,\ (\ (\ function\ ssaG\ )\ x\ )\ list\_of\_values\ ) \in\ (\ evalssa\ ssaG\ eval\_order\ )$
monos $evalssa\_mono\_helper\ evalssa\_mono\_helper2$

Fig. 4. Specification of the Evaluation of SSA Blocks with Additional Dependencies

$x \in base\ ssaG$

says that x must be a node in the graph. If these three preconditions are fulfilled, then x is included in the set *evalssa*. The thereby inductively defined set contains all pairs consisting of an element in the SSA partial order and of its result to which it is evaluated by applying its associated function to the values of its predecessors (which are also elements in the inductively defined set *evalssa ssaG eval_order*). The order in which elements are inductively included into the set *evalssa* corresponds to an evaluation order that respects the order relation in the original SSA partial order *ssaG* as well as the extra order dependencies contained in *eval_order*. Hence, *evalssa* defines an operational semantics for an SSA partial order and an additional set of dependencies that are to be respected during evaluation.

Based on these definitions, we have verified our main correctness theorem as stated below.

theorem $[\![sfpo\ (\ base\ (\ ssaG\ )\ ,\ rel\ ssaG\ )\ ;$
  $sfpo\ (\ sfpo\_union\ (\ base\ (\ ssaG\ )\ ,\ rel\ ssaG\ )\ eval\_order\ )\ ]\!]$
  $\Longrightarrow evalssa\ ssaG\ eval\_order\ =\ evalssa\ ssaG\ \{\}$

This theorem says that if we are given an SSA graph that corresponds to a strict finite partial order and if we insert the additional dependencies *eval_order* that do not destroy the property of the SSA graph of being partially ordered, then the original evaluation order in the SSA graph and the modified evaluation order which contains the original and the additional dependencies will return the same results. Speaking in the language of compilers, any sequence of instructions in the machine code that respects the data flow dependencies in the SSA basic blocks is a correct serialization.

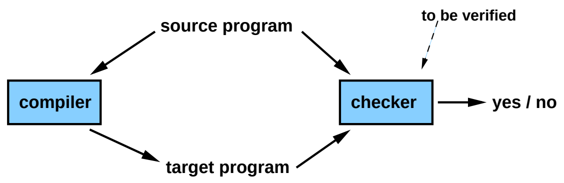
6 Comparison of the Formalization Alternatives

With the theories presented in Sections 3, 4, and 5, we have formally verifed the same results based on two different formalizations, namely on term graphs and on partial orders. Both proofs show that code generation for SSA basic blocks is correct if the data flow dependencies are preserved. If one is only interested in this result, then both formalizations are equally good. However, from a mathematical point of view, the two proofs are very different. There is an experience behind that many mathematicians know from their work. There are proofs which feel good because one has hit the right intuition. Even though one cannot generally define when a proof is good, one knows nevertheless most of the time exactly if a proof is indeed right, cf. also [1]. Let us also emphasize that the quality of implementation and proof decisions do not need to coincide. A certain implementation decision might be an excellent choice for obtaining an efficient running system while a proof with a formalization based on the same idea can be unnecessarily complicated.

We also made the experience that the choice of proof formalization is crucial. The second formalization based on partial orders as presented in the previous section fits to our intuitive proof idea and formalizes a general principle, namely that the transformation of a program must preserve its data dependencies. The second formalization is clearly superior to the first because it is shorter (only 600 lines of proof code, about 180 of them for the SSA specific part) and also more general: Note that most of the formalization in Section 5 is independent from SSA and that SSA only comes in in Subsection 5.3. By building up our proof on this general principle we will be able to reuse our proof for the verification of further transformations involving data flow dependencies. It is a challenge in general to find good abstractions in the area of formal verification because only then, verification cost can be brought down to an acceptable level.

7 Integration into Checker Approach

In recent years, program checking (also known as translation validation) has been established as the method of choice to ensure the correctness of compiler implementations: Instead of verifying a compiler, one only verifies its results. The correctness results presented in Sections 4 and 5 concern only the correctness of the code generation algorithm but not of its implementation. In this section, we show how these formally verified correctness results can be connected with the program checking approach in order to ensure that a given compiler implementation produces correct machine code.



The figure on the left-hand side demonstrates the principle of program checking. First the compiler computes the translated program. Then the independent checker evaluates a sufficient condition which classifies correct results. Our is_topsort and strict-finite-partial-order predicates defined in section 4 and 5 are such sufficient criterions for the correctness of the generated machine code for a given basic block. Its sufficiency has been formally verified by our theorems. So to check the correctness of the generated machine code, the checker checks if the topsort resp. sfpo criterion holds for the SSA basic block and the generated machine code. This check can be efficiently computed. With a checker implementing this check, we are able to connect the formal proof for the algorithmic correctness of code generation with a concrete compiler implementing it.

8 Related Work

Early work on formal correctness proofs for compilers [15] was carried out in the Boyer-Moore theorem prover considering the translation of the programming language Piton. Recent work has concentrated on transformations taking place in compiler frontends. [19] describes the verification of lexical analysis in Isabelle/HOL. The formal verification of the translation from Java to Java byte code and formal byte code verification was investigated in [22,14]. Further related work on formal compiler verification was done in the german Verifix project [8,9] focusing on correct compiler construction: [7] considers the verification of a compiler for a Lisp subset in the theorem prover PVS. The approach of proof-carrying code [17] is weaker than ours because it concentrates only on the verification of necessary but not sufficient correctness criteria. The approach of program checking has been proposed by the Verifix project [8] and has also become known as translation validation [21,18], recently also for loop transformations [13]. For an overview and for results on program checking in optimizing backend transformations cf. [10].

9 Conclusions and Future Work

We have presented a framework for the verification of code generation from SSA form. Our framework comprises two formalization and verification approaches. Our first approach formalizes SSA basic blocks as term graphs, whereas our second approach represents basic blocks as partial orders. Starting from general mathematical notions such as partial orders, acyclic graphs, and reachability, we have developed in our second approach a theory in Isabelle/HOL that formalizes data flow driven computations. In particular, we have verified that each transformation that only inserts new dependencies by keeping the original ones is correct. We have applied this general theory to SSA basic blocks and machine code generation. Thereby we have shown that each generated sequence of machine instructions is correct if the data dependencies of the original SSA form are preserved. However our second approach is by far more general than the first. It can and we plan to reuse it for verification purposes in areas where semantics is determined through partial orders, e.g. code generation and selection for parallel processors and loop transformations.

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 Section 5 are supposed to serve as basis. For many of these optimizations, it is necessary to move instructions between basic blocks. We are convinced that the specification and correctness proof stated in this chapter are a good basis to also verify such advanced algorithms.

References

[1] Martin Aigner and Günter M. Ziegler. *Proofs from THE BOOK*. Springer, 2004.

[2] Jan Olaf Blech and Sabine Glesner. A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL. In *Proc. der 3. Arbeitstagung Programmiersprachen (ATPS), 34. Jahrestagung der Gesellschaft für Informatik*. Lecture Notes in Informatics, 2004.

[3] Jan Olaf Blech, Sabine Glesner, Johannes Leitner, and Steffen Mülling. Some Theorems on Data Dependencies using Partial Orders, 2004. Internal Report, University of Karlsruhe.

[4] Jan Olaf Blech. Eine formale Semantik für SSA-Zwischensprachen in Isabelle/HOL. Diplomarbeit (Master's Thesis), University of Karlsruhe, 2004.

[5] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[6] 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 TOPLAS*, 13(4):451–490, October 1991.

[7] Axel Dold, Friedrich W. von Henke, and Wolfgang Goerigk. A Completely Verified Realistic Bootstrap Compiler. *International Journal of Foundations of Computer Science*, 14(4):659–680, 2003.

[8] W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F.W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The Verifix Approach. In P. Fritzson, editor, *Poster Session of CC'96*. IDA Technical Report LiTH-IDA-R-96-12, Linkoeping, Sweden, 1996.

[9] Sabine Glesner, Gerhard Goos, and Wolf Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). *it - Information Technology*, 46:265–276, 2004. Print ISSN: 1611-2776.

[10] Sabine Glesner. Using Program Checking to Ensure the Correctness of Compiler Implementations. *Journal of Universal Computer Science (J.UCS)*, 9(3):191–222, March 2003.

[11] Sabine Glesner. An ASM Semantics for SSA Intermediate Representations. In *Proc. 11th Int'l Workshop on Abstract State Machines*, 2004. Springer, Lecture Notes in Computer Science.

[12] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, ed., *Specification and Validation Methods*. Oxford University Press, 1995.

[13] B. Goldberg, L. Zuck, and C. Barrett. Into the Loops: Practical Issues in Translation Validation for Optimizing Compilers. In *Proc. Workshop Compiler Optimization meets Compiler Verification (COCV 2004)*, 2004. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS).

[14] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298:583–626, 2003.

[15] J. S. Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[16] Steven S. Muchnick. *Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[17] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997.

[18] George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[19] Tobias Nipkow. Verified Lexical Analysis. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*. Springer, Lecture Notes in Computer Science, Vol. 1479, 1998. Invited talk.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Lecture Notes in Computer Science, Vol. 2283, 2002.

[21] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, 1998. Springer, Lecture Notes in Computer Science, Vol. 1384.

[22] Martin Strecker. Formal Verification of a Java Compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, pages 63–77. Springer, Lecture Notes in Computer Science, Vol. 2392, 2002.