A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL

Jan Olaf Blech and Sabine Glesner

Abstract: Optimizations in compilers are the most error-prone phases in the compilation process. Since correct compilers are a vital precondition to ensure software correctness, it is necessary to prove their correctness. In this paper, we develop a formal semantics for static single assignment (SSA) intermediate representations and prove formally within the Isabelle/HOL theorem prover that a relatively simple form of code generation preserves the semantics of the transformed programs in SSA form. This formal correctness proof does not only verify the correctness of a certain class of code generation algorithms but also gives us a sufficient, easily checkable correctness requirement characterizing correct compilation results obtained from implementations (i.e. compilers) of these algorithms.

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 a theorem prover. 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 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 preparatory work [Gl04], we have shown that SSA semantics can be captured elegantly and adequately with abstract state machines [Gu95]. Based on this work, we develop a formal SSA semantics within the theorem prover Isabelle/HOL [NPW02]. The imperative semantics transfers control flow from one basic block to its successor basic block, i.e. the current state is characterized by the currently executed basic

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

block and by the results computed by the previously executed blocks. Within basic blocks, SSA computations are purely data-flow driven. These computations are typically represented by acyclic directed graphs representing the data dependences. In our formalization, we have represented these graphs by *termgraphs* [BN98]. Termgraphs represent acyclic graphs by duplicating common subexpressions. To keep track of the 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 termgraphs. Based on this formalization, we define a formal semantics for SSA basic blocks by stating a function that evaluates term graphs. Our specification of SSA semantics is well-suited to formally prove correctness of code generation algorithms. In this paper, we formally prove the correctness of a relatively simple code generation algorithm. Thereby we prove that every topological sorting of data flow dependencies within a basic block is a correct code generation order because then the generated machine program preserves the data flow dependencies of the SSA program. Furthermore, we point out how this proof can be extended to capture also more complex optimization strategies during code generation. In our work, we have used the Isabelle/HOL system [NPW02] to specify the SSA language and to carry out our correctness proof. As a by-product, this formal proof yields an easily checkable criterion classifying correct compilation results. This criterion can easily be integrated into the well-established approach of program result checking [Gl03] (also known as translation validation [PSS98]) typically used to ensure correctness of compiler results.

This paper is organized as follows: First, we introduce SSA intermediate representations in section 2. Then we define their formal semantics within Isabelle/HOL, cf. section 3. Afterwards, in section 4, we formally prove the correctness of code generation. In section 5, we show how this formal correctness proof can be connected with the program checking approach used to verify the correctness of individual translated programs. We conclude this paper with a discussion of related work in section 6 and conclusions and aspects of future work in section 7.

2 SSA - Based Intermediate Languages

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 block[2] 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.

[2] 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.

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*, $x = \phi(x_1, \ldots, x_n)$, 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. n is the number of predecessors of the basic block. Programs can easily be transformed into SSA representation, cf. [Mu97], e.g. during a tree walk through the attributed syntax tree. The standard transformation algorithm subscripts each variable. At join points, ϕ nodes sort out multiple assignments to a variable which correspond to different control flows through the program.

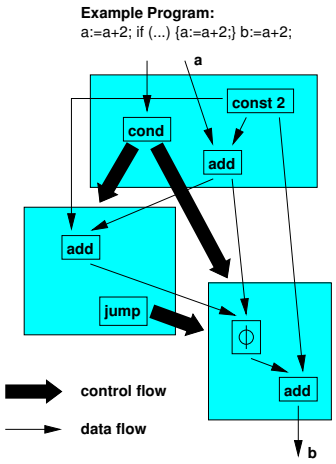

Figure 1: SSA Representation

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. 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 in the previously executed basic blocks.

Memory accesses need special treatment. In the functional store approach [St95], memory read/write nodes are considered as accesses to fields of a global state variable *memory*. A write access modifies this global variable *memory* and requires that the outcome of this 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 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 to several independent memories.

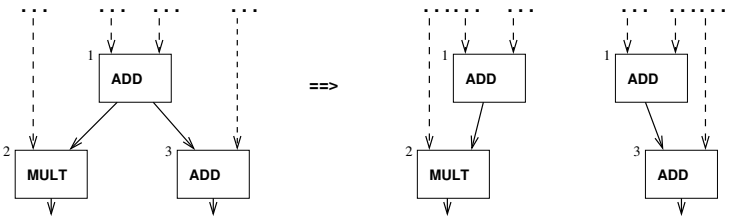

Figure 2: SSA DAG $\Longrightarrow$ SSA Tree

3 A Formal SSA Semantics in Isabelle/HOL

In this section we describe the specification of SSA based intermediate languages within the Isabelle/HOL system: First, in subsection 3.1, we formalize the data flow within basic blocks. Then, in subsection 3.2, we describe the global control and data flow.

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 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 we have represented such a DAG by transforming it into an equivalent set of trees by duplicating shared subterms, cf. Figure 2. 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 $SSATree =$
$\quad CONST\ value\ identifier\ |$
$\quad PHI\ phiargs\ value\ identifier\ |$
$\quad NODE\ operat\ SSATree\ SSATree\ value\ identifier\ |$
$\quad LOAD\ SSATree\ SSATree\ value\ identifier\ |$
$\quad STORE\ SSATree\ SSATree\ SSATree\ memory\ identifier\ |$
$\quad MEMORY\ memory\ identifier$

Nodes represent constants, ϕ-nodes with argument lists, arithmetic operations, and memory accesses. Each node has two associated numbers assigned to it, the ***value*** number representing the result of the corresponding operation and the ***identifier*** number enabling identification of identical subterms. Memory operations are specified according to the functional store approach [St95], cf. section 2. $MEMORY\ memory\ identifier$ represents the state of memory at the beginning of the evaluation of a given basic block. $LOAD$ and $STORE$ are the usual operations which load and store values from and in memory. Result of the load operation is the fetched value, result of the store operation is the updated memory. SSA basic blocks are evaluated with the evaluation function $eval\_tree$ which is defined inductively on SSA trees. Since memory operations are formalized functionally, they can be defined in the same format as the purely functional operations:

consts
$eval\_tree :: \text{“}SSATree \Rightarrow SSATree''$
primrec
$''eval\_tree\ (LEAF\ val\ ident) = (LEAF\ val\ ident)''$
. . . . .
$''eval\_tree\ (NODE\ operat\ tree1\ tree2\ val\ ident)\ =$
$(NODE\ operat\ (eval\_tree\ tree1)\ (eval\_tree\ tree2)\ (operat$
$(get\_ssatree\_val\ (eval\_tree\ tree1))$
$(get\_ssatree\_val(eval\_tree\ tree2)))$
$ident)''$
. . . . .

3.2 Formal Semantics for the Global Control and Data Flow

An SSA program is formalized as a list of basic blocks:

datatype
$BASICBLOCK =$
$NEW\ identifier\ identifier\ \text{“}identifier \times nat''\ \text{“}identifier \times nat''\ \text{“}SSATree\ list''$

Each basic block carries four pieces of information which integrates it into the global control and data flow, specified with these five fields:

1. $identifier$	the value number that determines the next basic block
2. $identifier$	the value number that determines the memory state for the next basic block
3. $identifier \times nat$	successor target 1 and its rang
4. $identifier \times nat$	successor target 2 and its rang
5. $SSATree\ list$	list of SSATrees containing the operations of the basic block

In our formalization, a basic block b can have only 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 characterising the successor block. nat specifies its rang and defines the selection of the arguments in the ϕ nodes in b': If rang is i, then the ith argument in the argument list of a ϕ node in b' is chosen.

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, by the current state of memory, and by the currently executed basic block. Therefore we specify:

- a table of values — formalized as a function ($identifier \Rightarrow value$) indexed by value number
- a memory state — ($identifier \Rightarrow value$), indexed by memory address
- no. of current basic block

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

- it assigns each ϕ-node its value
- it assigns the memory function ($identifier \Rightarrow value$) to each memory node
- 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 collects the memory state for the next basic block from a distinct node
- it determines the id of the next basic block with a 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 intermediate languages. For a complete specification with all details, we refer to [Bl04].

4 Correctness of Code Generation

In this section, we consider a relatively simple code generation algorithm and prove part of its correctness by showing that it preserves the obervable 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. In subsection 4.1, we introduce the simple machine language. Subsection 4.2 defines the criterion for topological sorting. In subsection 4.3, we summarize the Isabelle/HOL proof.

4.1 Semantics of the Machine Language

Machine code is formalized as a list of CodeElements which operate on values stored in memory. Memory is specified as a function "$(nat \Rightarrow value)$" that maps $identifiers$

to the contents of storage cells. Since we concentrate on the correct translation of basic blocks, it is sufficent to work with the following machine language:

datatype $CodeElement = L\ value\ identifier\ |$
$N\ operator\ identifier\ identifier\ identifier$

The "$L\ value\ identifier$"-element has the following semantics: store $value$ at storage 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 memory:

eval_codelist :: $''CodeList \Rightarrow (nat \Rightarrow value) \Rightarrow (nat \Rightarrow value)''$

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 that maps an SSATree(node) to a code element. Our formalization of topological sortings, the predicate is_topsort, covers the following aspects:

- 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 to ce_ify b in the code_list.
- Each Element in the code list has a unique id.

A detailed description of the Isabelle/HOL specification defining these requirements can be found in [Bl04]. As example, the first requirement is formalized in Isabelle/HOL by:

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

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.

theorem main theorem**:**
$''(\forall\ clist.\ ((is\_topsort\ clist\ tree) \longrightarrow$
$(\forall\ t.(t \in (eval\_tree\ tree)) \longrightarrow$
$(\exists\ ident\ val.(val = (eval\_codelist\ clist(\lambda\ a.(Eps(\lambda\ a.\ False)))\ ident) \wedge$
$(val = get\_ssatree\_val\ t) \wedge (ident = get\_ssatree\_id\ t)))))''$

We prove this theorem by an induction over the SSATree $tree$. In the base case we show:

> Assume is_topsort (clist,LEAF val ident) $\Longrightarrow$ val. Then the result of $LEAF val\ ident$ is also computed by the machine program and is available under value number $ident$ after the execution of $clist$. To prove this, we need an auxiliary lemma stating that the is_topsort criterion (in particular its parts tops1, tops2 and tops4) is only satisfied if clist has the form [L val ident]:
> **Auxiliary lemma** $endlist\_a$:
> "$[|tops1\ (l@[x])\ T; tops2\ (l@[x])\ T; tops4\ (l@[x])\ T|] \Longrightarrow\ x = ce\_if y\ T''$
> With this lemma, the proof of the induction base case is trivial. (Note that LEAF can be either be a CONST or a PHI node).

Proving the induction-step is more difficult. We have the following 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.

In our proof, we have skolemized the $\forall$-quantified variables list' and list" in the induction assumptions by instantiating them with "proj (list,kid1)" and "proj (list,kid2)". "proj (CodeElement list $\Rightarrow$ SSATree $\Rightarrow$ CodeElement list)" is a function that maps all elements from the input CodeElement list that have a corresponding element in the SSATree to the output CodeElement list. In our proof we have defined the proj function via its properties. From the induction hypotheses and the characteristics of the proj function, we can derive that every value that gets calculated in kid1 and kid2 will be calculated in the CodeElement list $list$.

For every subtree t of $tree$ we have to show that its values will be calculated in the code list. We prove this by the following case distinction:

1. t is subtree of kid1
2 t is subtree of kid2
3. t is the root node: $tree$

Cases 1 and 2 can be derived from the induction hypotheses and the characteristics of the proj function. In order to prove case 3, we show that for every topological sorted list of a tree the last element corresponds to the root node. From the fact that every child node is correctly evaluated in the CodeElement list we derive that the root node will be evaluated correctly as well.

The entire proof has been carried out within the Isabelle/HOL system. Our proof verifies 45 lemmas and the main theorem. In total, our proof theory file contains about 885 lines of proof code.

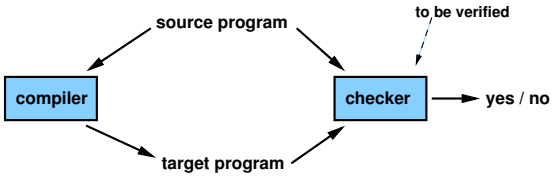

Figure 3: Program Checking with Certificates

5 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 result presented in section 4 concerns only the correctness of the code generation algorithm but not of its implementation. In this section, we show how this formally verified correctness result can be connected with the program checking approach in order to ensure that a given compiler implementation produces correct machine code.

Figure 3 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 predicate defined in section 4 is a sufficient criterion for the correctness of the generated machine code for a given basic block. Its sufficiency has been formally verified by our main theorem. So to check the correctness of the generated machine code, the checker checks if the topsort 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.

6 Related Work

Related work on formal correctness proofs for compilers has concentrated on transformations taking place in compiler frontends. [Ni98] describes the verification of lexical analysis within the Isabelle/HOL system. [WW97] partly proves the correctness of lexical and syntactic analysis in compilers; because of the complexity of these proofs, she did not succeed in completing them. The formal verification of the translation from Java to Java byte code and formal byte code verification have been investigated in [St02, KN03]. Further related work on formal compiler verification has been done in the german Verifix project [GDG$^+$96] which focuses on correct compiler construction: In [DV01, DvHVG02] a compiler for a Lisp subset has been partially verified using the theorem prover PVS.

The approach of program checking has been proposed by the Verifix project [GDG$^+$96] and has also become known as translation validation [PSS98]. For an overview and for results on program checking in optimizing backend transformations cf. [Gl03].

7 Conclusion

In this paper, we have presented a formal semantics for SSA intermediate representations in a simple and elegant way within the theorem prover Isabelle/HOL. Thereby we represented common subexpressions in basic blocks by termgraphs. Based on this formalization, we verified the correctness of a relatively simple code generation algorithm by proving that the semantics of the translated programs is preserved. In particular, we proved that every topological sorting of the operations in a basic block is a correct code generation order. We have carried out this proof in Isabelle/HOL. Thereby we have demonstrated that our SSA specification is a suitable basis for correctness proofs. We also showed how to connect this formal proof with a concrete compiler implementation by exploiting the approach of program checking.

In ongoing work, we are using this specification to prove the correctness of data flow analyses (e.g. live variables analysis and dead code elimination). In future work, we also want to extend the machine language to include very long instruction words (VLIW), predicated instructions, and speculative execution. This also implies that we need to consider more advanced code generation algorithms which aggressively explore the inherent data dependencies to generate efficient code for such architectures. We are convinced that the specification and correctness proof stated in this paper are a good basis to also verify such advanced algorithms.

References

[Bl04] Blech, J. O. Eine formale Semantik für SSA-Zwischensprachen in Isabelle/HOL. Diplomarbeit (Master's Thesis), Betreuung (advisor): Dr. Sabine Glesner, Universität Karlsruhe, Fakultät für Informatik. 2004.

[BN98] Baader, F. und Nipkow, T.: *Term Rewriting and All That*. Cambridge University Press. 1998.

[CFR+91] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., und Zadeck, F. K.: 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] Dold, A. und Vialard, V.: A Mechanically Verified Compiling Specification for a Lisp Compiler. In: *Proceedings of the 21st Conference on Software Technology and Theoretical Computer Science (FSTTCS 2001)*. S. 144–155. Bangalore, India. December 13-15 2001. Springer Verlag, Lecture Notes in Computer Science, Vol. 2245.

[DvHVG02] Dold, A., von Henke, F. W., Vialard, V., und Goerigk, W.: A Mechanically Verified Compiling Specification for a Realistic Compiler. Ulmer Informatik-Berichte 02-03. Universität Ulm, Fakultät für Informatik. 2002.

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

[Gl03] Glesner, S.: Using Program Checking to Ensure the Correctness of Compiler Implementations. *Journal of Universal Computer Science (J.UCS)*. 9(3):191–222. March 2003.

[Gl04] Glesner, S.: An ASM Semantics for SSA Intermediate Representations. In: *Proceedings of the 11th International Workshop on Abstract State Machines*. Halle, Germany. May 2004. Springer Verlag, Lecture Notes in Computer Science.

[Gu95] Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In: Börger, E. (Hrsg.), *Specification and Validation Methods*. S. 231–243. Oxford University Press. 1995.

[KN03] Klein, G. und Nipkow, T.: Verified Bytecode Verifiers. *Theoretical Computer Science*. 298:583–626. 2003.

[Mu97] Muchnick, S. S.: *Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc. 1997.

[Ni98] Nipkow, T.: Verified Lexical Analysis. In: Grundy, J. und Newey, M. (Hrsg.), *Theorem Proving in Higher Order Logics*. S. 1–15. Springer Verlag, Lecture Notes in Computer Science, Vol. 1479. 1998. Invited talk.

[NPW02] Nipkow, T., Paulson, L. C., und Wenzel, M.: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Lecture Notes in Computer Science, Vol. 2283. 2002.

[PSS98] Pnueli, A., Siegel, M., und Singermann, E.: Translation validation. In: Steffen, B. (Hrsg.), *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*. S. 151–166. Lisbon, Portugal. April 1998. Springer Verlag, Lecture Notes in Computer Science, Vol. 1384.

[St95] Steensgaard, B.: Sparse Functional Stores for Imperative Programs. In: *The First ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA*. S. 62–70. 1995.

[St02] Strecker, M.: Formal verification of a Java compiler in Isabelle. In: *Proc. Conference on Automated Deduction (CADE)*. volume 2392 of *Lecture Notes in Computer Science*. Springer Verlag. 2002.

[WW97] Weber-Wulff, D.: *Contributions to Mechanical Proofs of Correctness for Compiler Front-Ends*. PhD thesis. Christian-Albrecht-Universitt zu Kiel. Kiel, Germany. April 1997. available as technical report "Bericht Nr. 9707".