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

Jan Olaf Blech and Sabine Glesner

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe, 76128 Karlsruhe, Germany

Abstract: Optimizations in compilers are the most error-prone phases in the compilation process. Since correct compilers are a vital precondition for software correctness, it is necessary to prove their correctness. 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 criterion characterizing correct compilation results obtained from implementations (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 previous 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. The imperative semantics transfers control flow from one basic block to its successor block,

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

i.e. the current state is characterized by the currently executed basic 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 in a basic block is a correct code generation order because then the generated machine program preserves the data flow dependencies. Furthermore, we point out how this proof can be extended to also capture 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.

2 SSA - Based 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 [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 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 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. n is the number of predecessors of this block. Programs can easily be transformed into SSA form, cf. [Mu97], 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 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. 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.

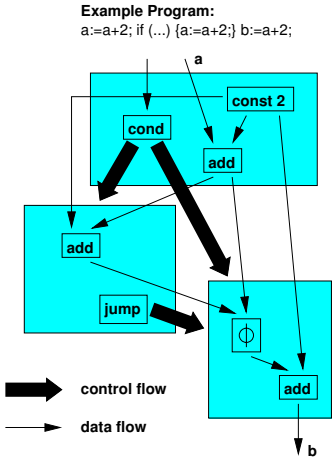

Figure 1: SSA Representation

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.

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 since the same semantic description can be used for accesses to only a single as well as to several independent memories.

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, con-

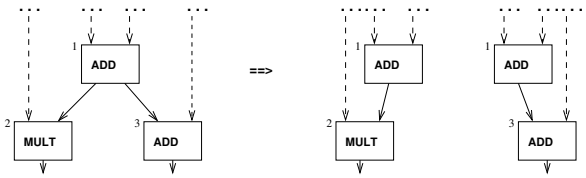

Figure 2: Transforming SSA DAGs into SSA Trees

stants, 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 $SSATree = CONST\ value\ identifier \mid PHI\ phiargs\ value\ identifier \mid$
$NODE\ operator\ SSATree\ SSATree\ value\ identifier \mid$
$LOAD\ SSATree\ SSATree\ value\ identifier \mid$
$STORE\ SSATree\ SSATree\ SSATree\ memory\ identifier \mid$
$MEMORY\ memory\ identifier$

Nodes represent constants, ϕ-nodes with argument lists, arithmetic operations, and memory accesses. Each node has two associated numbers assigned to it, the *value* representing the result of the corresponding operation and its $identifier$. Memory accesses 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; $identifier$ being the identifier of this constant (wrt. a basic block) function. $LOAD$ and $STORE$ are the usual operations which load and store values from and in memory. Both get the address to be loaded from or stored to, resp., as well as the current memory and, in case of the store operation, the value to be stored as operands which are $SSATrees$. 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.

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.

```
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)"        ...
```

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 five pieces of information which integrates it into the global control and data flow:

```
datatype BASICBLOCK =
   NEW identifier identifier "identifier × nat" "identifier × nat" "SSATree list"
```

1. *identifier*	the value number that determines the successor basic block
2. *identifier*	the value number that determines the memory state for the successor basic block
3. *identifier* × *nat*	successor target 1 and its rank
4. *identifier* × *nat*	successor target 2 and its rank
5. *SSATree list*	list of SSATrees containing 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, 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 numbers
- a memory state — ($identifier \Rightarrow value$), indexed by memory addresses
- 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 assigns the initial memory constant ($identifier \Rightarrow value$) to each initial 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 the corresp. distinct memory node
- it determines the successor basic block with the corresponding 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. 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. For simplicity, we do not consider memory operations in this paper. Furthermore, since we prove correctness of code generation for individual basic blocks, we can treat PHI 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 $(identifier \Rightarrow value)$ 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 $CodeElement = L\ value\ identifier \mid N\ operator\ identifier\ identifier\ identifier$

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:

$eval\_codelist\ ::\ ''CodeList \Rightarrow (identifier \Rightarrow value) \Rightarrow (identifier \Rightarrow value)''$

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[2] that maps an SSATree (node) to a code element

[2] ce_ify stands for *CodeElementify*.

($SSATree \Rightarrow CodeElement$). Our formalization of topological sortings, formally defined by 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 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 [Bl04]. As example, the first requirement is formalized in Isabelle/HOL by:

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

is_in_cl ($CodeElement \Rightarrow CodeElement\ list \Rightarrow bool$) is a predicate which holds if $CodeElement$ is contained in $CodeElement\ list$. 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**:**

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

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

Proof of Base Case: We need to show that if $is\_topsort\ clist\ (LEAF\ val\ ident)$ holds, then the result of $LEAF val\ ident$ is also computed by the machine program and is available under value number $ident$ after execution of $clist$. Therefore we need a lemma stating that the is_topsort criterion is only satisfied if clist has the form $[L\ val\ ident]$:

Auxiliary lemma: $''((is\_topsort\ clist\ (LEAF\ val\ ident)) \longrightarrow (clist = [L\ val\ ident]))''$

With this lemma, the proof of the induction base case is trivial. (Note that $LEAF$ can either be a $CONST$ or a PHI node).

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 $\forall$-quantified variables $list'$ and $list''$ in the induction assumptions by instantiating them with $proj\ list\ kid1$ and $proj\ list\ kid2$. The function $(proj\ ::\ ''CodeElement\ list \Rightarrow SSATree \Rightarrow CodeElement\ list'')$ 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 have to show that its values will be 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. ■

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

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.

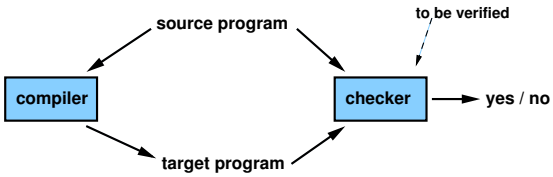

Figure 3: Program Checking

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

Early work on formal correctness proofs for compilers [Mo89] 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 front-ends. [Ni98] 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 [St02, KN03]. Further related work on formal compiler verification was done in the german Verifix project [GDG+96] focusing on correct compiler construction: [DvHG03] considers the verification of a compiler for a Lisp subset in the theorem prover PVS. The approach of proof-carrying code [Ne97] 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 [GDG+96] and has also become known as translation validation [PSS98, Ne00]. 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 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/dead code elimination). In future work, we want to extend the machine language to include very long instruction words (VLIW), predicated instructions, and speculative execution. This 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. In addition, we are experimenting with alternative formalisms which represent basic blocks 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. It seems that both formalisms, the one with partial orders and the one presented in this paper, have their special advantages and disadvantages, depending on the proof goals.

Acknowledgments: The authors would like to thank the anonymous reviewers for their helpful comments. This work was supported by a research grant within the “Eliteförderprogramm für Postdoktoranden der Landesstiftung Baden-Württemberg”.

References

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

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

[CFR+91] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and 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.

[DvHG03] Dold, A., von Henke, F. W., and Goerigk, W.: A Completely Verified Realistic Bootstrap Compiler. *Int'l Journal of Foundations of Comp. Science*. 14(4):659–680. 2003.

[GDG+96] Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A., von Henke, F., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H., and Zimmermann, W.: Compiler Correctness and Implementation Verification: The Verifix Approach. In: *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 Comp. Science (J.UCS)*. 9(3):191–222. March 2003.

[Gl04] Glesner, S.: An ASM Semantics for SSA Intermediate Representations. In: *Proc. 11th Int'l Workshop on Abstract State Machines*. May 2004. Springer, LNCS Vol. 3052.

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

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

[Mo89] Moore, J. S.: A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*. 5(4):461–492. 1989.

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

[Ne97] Necula, G. C.: Proof-Carrying Code. In: *Proc. Symposium on Principles of Programming Languages (POPL'97)*. pages 106–119. 1997.

[Ne00] Necula, G. C.: Translation Validation for an Optimizing Compiler. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. pages 83–94. Vancouver, British Columbia, Canada. May 2000.

[Ni98] Nipkow, T.: Verified Lexical Analysis. *Theorem Proving in Higher Order Logics*. pages 1–15. Springer, LNCS, Vol. 1479. 1998. Invited talk.

[NPW02] Nipkow, T., Paulson, L. C., and 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., and Singerman, E.: Translation validation. *Proc. Tools and Algorithms for the Construction and Analysis of Systems*. 1998. Springer, LNCS, Vol. 1384.

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

[St02] Strecker, M.: Formal verification of a Java compiler in Isabelle. *Proc. Conference on Automated Deduction (CADE)*. Springer, LNCS, Vol. 2392. 2002.