Certifying Code Generation Runs with Coq: A Tool Description

Jan Olaf Blech[1]

University of Kaiserslautern, Germany

Benjamin Grégoire[2]

INRIA Sophia Antipolis, France

Abstract

In this tool description paper we present a certifying code generation phase. Our code generation phase takes intermediate language programs and translates them into MIPS assembler code. Each time our code generation is invoked a proof script is emitted. This proof script is used as a certificate to guarantee the correctness of the code generation run. It is checked in the Coq theorem prover. Once this has been successfully done we can be sure that the code generation has been carried out correctly. Checking the generated proof scripts has turned out to be a bottleneck of certifying compilation. This paper is based on an implementation which uses – among other techniques – checker predicates to overcome this bottleneck. These are predicates formalized in an executable way that can be easily evaluated by the Coq theorem prover to speed up the certificate checking process.
This paper presents the certifying code generation phase introduced in [5] and focuses on its implementation.

Keywords: Translation Validation, Certifying Compilation, Theorem Proving, Coq, Tool Description

1 Overview

The technique of compilers that emit a certificate thereby guaranteeing the correctness of distinct compilation runs *certifying compilers* (cp. Figure 1) is our methodology of choice to achieve a trustable code generation: For each compilation(-phase) run source and target program are passed to a checker – the Coq theorem prover, in our case – together with a certificate – the proof script – to check whether the compilation run has been done correctly. The proof script is automatically generated by the compiler and contains lists of tactic applications and other hints to guide the theorem prover through the checking process.

[1] Email: blech@informatik.uni-kl.de

[2] Email: Benjamin.Gregoire@sophia.inria.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

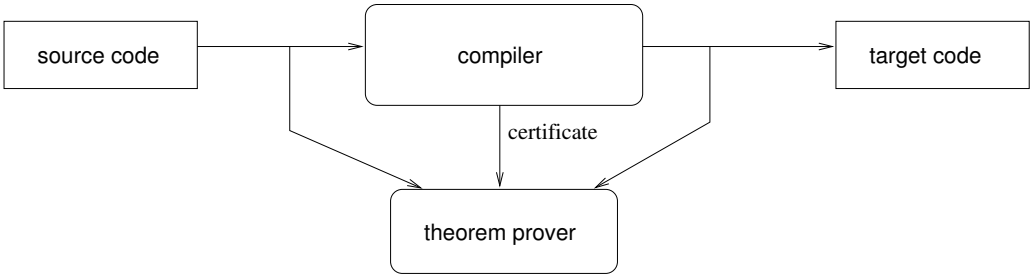

Fig. 1. Certifying Compiler

Certifying compilers have several advantages over *certified compilers*. These are compilers where the compilation algorithm and its implementation have been verified thereby guaranteeing correctness of compilation runs [3].

- First, the issue of implementation correctness can be completely avoided.
- Second, the technique provides a clear interface between compiler producer and user. No internal details of the compiler have to be given to the compiler user in order to guarantee the correctness of a compiler run.
- Third, certifying compilers allow us to abstract from implementation details. This frees us from reverifying the compiler once an aspect of implementation changes slightly.
- Fourth, the generation of certificates is often easier than the verification of a compilation algorithm. We do not have to consider very unlikely input. Thus, we can give up completeness to make the verification task easier.

However, certificates have to be generated and checked for each compilation run.

In this tool description paper we present a certifying code generation phase. Certificates are proof scripts for use with a theorem prover. We use the Coq [24] theorem prover as proof checker. A higher order theorem prover like Coq as certificate checker has the following characteristics which we have used in our certifying code generation phase:

- It allows us to formalize the operational semantics of the involved languages in a convenient and human readable way.
- We have formalized and use in our correctness proofs an explicit criterion stating correctness of compilation based on higher-order logic.
- Higher-order theorem provers provide human readable feedback in case a proof fails. This makes it easy to debug the compiler or the proof script generator.
- Higher-order theorem provers are relatively slow compared to first-order theorem provers or similar certificate checking tools.

Thus, our certifying code generation features explicit syntax and semantics definitions of the involved languages. We have established a correctness criterion under which we regard a compiler run as conducted correctly. This notion of correctness is based on simulation between the involved programs. We require generated output traces to be the same in both source and target program during their execution.

[3] See [14] for a refined classification of certifying and certified compilers.

This notion adequately captures the compilation correctness of terminating and non-terminating programs.

We have proposed our methodology of certifying compilers in [22,8] and applied it to the code generation phase in different implementations [6,2,5]. This paper describes the structure and implementation – one focus is on the generation of the proof scripts – of the certifying code generation described in [5]. The paper [5] focuses on the semantics and compilation correctness definitions, the correctness proofs and the process of proving compilation runs correct.

Related Approaches

The following approaches use a certifying methodology to guarantee correctness or other properties of compilation.

- In the translation validation approach [21,1,27] the compiler is regarded as a black box with at most minor instrumentation. For each compiler run, source and target program are passed to a separate checking unit comprising an analyzer generating proofs. These proofs are checked with a proof checker. A translation validation approach and implementation for the GNU C compiler is described in [18]. The analyzer generating the proofs in translation validation corresponds to our certificate generator. In contrast to translation validation our approach is based on a general higher-order proof assistant as for checking the certificates and explicitly formalized semantics. Furthermore, we use more information to generate the proof scripts from the compiler.
- A translation validation checker (called validator) has been formally verified in [25]. Here – like in our work – correctness is based on formalized semantics. The validator is used for verifying instruction scheduling. It is generated out of a verified Coq specification.
- Credible compilation [23] is another approach for certifying compilers. Credible compilation largely uses instrumentation of the compiler to generate proof scripts. Like translation validation and in contrast to our work credible compilation is not based on a explicitly formalized semantics.
- Proof carrying code [17] is a framework for guaranteeing that certain requirements or properties of a compiled program are met, e.g. type safety or the absence of stack overflows. These are necessary conditions that have to be fulfilled in a correctly compiled program. In contrast to proof carrying code we require in our work a comprehensive notion of compilation correctness. In [15] a compiler generating certificates for the proof carrying code approach that guarantees that target programs are type and memory safe is described. The clear separation between the compilation infrastructure and the checkable certificate is realized in our approach as well.

Certified compiler techniques are especially relevant to the logical framework of our approach. In [14], the algorithms for a sophisticated multi-phase compiler back-end are proved correct within the Coq theorem prover. To achieve a trusted implementation of the algorithm, it is exported directly from the theorem prover to program code. A similar approach based on Isabelle/HOL is presented in [11]. The verifica-

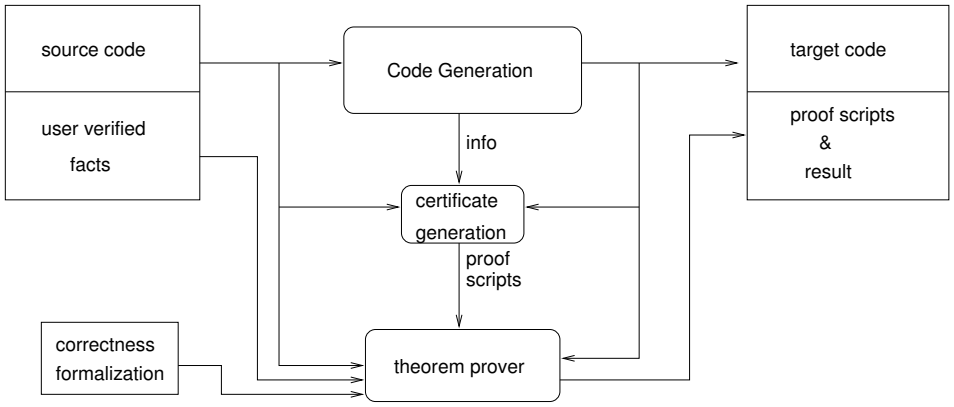

Fig. 2. Overview of Our Certifying Code Generation

tion of an optimization algorithm is described in [4]; it uses an explicit simulation proof scheme for showing semantical equivalence. In [13] a specification language is described for writing program transformations and their soundness properties. The properties are verified by an automatic theorem prover. Important techniques and formalisms for compiler result checkers, decomposition of compilers, notions of semantical equivalence of source and target program as well as stack properties were developed in the Verifix project [9,10,26] and in the ProCoS project [7]. The development of a formally verified compiler for a C subset is part of the Verisoft project focusing on pervasive formal verification of computer systems [12].

Overview

The rest of the paper is structured as follows: We give an overview on our certifying code generation phase in Section 2. Its implementation focusing on the generation of the proof scripts is described in 3. A performance evaluation is given in Section 4. In Section 5 a conclusion and ideas for future work are presented.

2 Our Certifying Code Generation Phase

Our code generation phase compiles intermediate language code into MIPS [20] assembler code. Our intermediate language consists of arithmetic expressions, (array-) variable assignments, (un)conditional branches, a print statement, and (potentially recursive) procedure call and return statements. Our MIPS language comprises basic arithmetic operations, shift operations, and branch instructions. Instructions for handling outputs and procedure calls are provided.

The overall architecture of our code generation phase is shown in Figure 2. Intermediate language programs are passed to the actual code generator. It comprises the following steps in order to generate MIPS assembler code: In a first step memory locations are determined for local and global variables and register allocation is done. We select MIPS instructions next and finally resolve jump targets. The code generation emits some information on the programs that are compiled. These are passed to the certificate generator which is well separated from the rest of the compiler for generating the proof scripts. The information provided by the compiler has to be computed anyway and comprises the variables used in the intermediate

language code, a mapping of variables to memory locations and registers (*variable mapping*), and a relation of intermediate language statements and corresponding pieces of MIPS code (*program counter relation*). The proof scripts prove that the MIPS code is a correct translation of the intermediate language code and are passed to the theorem prover. In addition to performing the code and certificate generation we emit theorem prover representations for intermediate language code and MIPS code, which are given to the theorem prover, too. We allow the use of facts that users have proved on the source code to ease some verification tasks or allow us to abandon some checks like index out of bounds checks in variable accesses during the checking process of the proof scripts. These facts are optional. Neither our code generation or the correctness proofs do depend upon them. It may be advantageous to keep the proof scripts so that third party users of the code can ensure themselves of the correctness of a compilation. Furthermore, the formalization of our notion of correctness is passed to the theorem prover before it starts to check the code generation run.

Note, that the actual code generation implementation, but also the generation of the proof scripts does not belong to the trusted computing base of certifying compilation. This means if we generate false proof scripts we will not be able to prove the corresponding code generation run correct. In this case we will discard the results of the compilation run although – if just the proof generation mechanism failed – it might have been correct.

Structure of the Proof Scripts

The proofs contained in the generated proof scripts can be structured into three phases:

- Prerequisites and properties concerning the register and memory allocation are ensured in a *first phase*. Most notably the injectivity of the mapping between variables and memory/register location is proved. Thus, no two variables share the same memory location at a distinct point of time.
- Our notion of compilation correctness is based on the equivalence of execution traces of the involved programs. Thus, we partition possible execution steps into a finite number of classes. We prove these classes of possible execution steps correct in a *second phase*.
- Finally, we put everything together to achieve an overall correctness proof in a *third phase*.

The time for checking the certificates is the bottleneck of certifying compilation. Our code generation phase realizes some techniques to overcome this obstacle. Some of the improvements are achieved by using a checker predicate in the first phase of our generated proof scripts. This is a predicate formalized in an executable way within the Coq theorem prover. Since we did prove our checker predicate equivalent to a classical, non-executable specification we use it instead of the non-executable specification in our generated proof scripts and gain speed improvements.

3 Implementation

So far we have implemented two different versions of certifying code generation:

- An earlier Isabelle/HOL [19] based version of our certifying code generation phase described in [6].
- The Coq version of our certifying code generation phase [5] that is described in this paper. In addition to Coq as the theorem prover it features some instrumentation with checker predicates and language extensions.

In this section we describe our implementation of a certificate generator for code generation (some of this work is described in [3]). Furthermore, we describe the other appliances for checking these certificates in the Coq theorem prover.

As pointed out in Section 2 the certificate generator uses information provided by the compiler comprising sets of variables used in the intermediate language, the variable mapping, and the program counter relation.

3.1 Program Representation Generation

Before generating the proof scripts and the theorem stating code generation run correctness we generate theorem prover representations of intermediate language and MIPS code. In addition, we generate representations of the variable sets, and formalizations of the program counter relation, and variable mapping to be used by the theorem prover. Apart from the theorem prover and hardware and operating system, the generation of the representations of intermediate language and MIPS code are the only parts in a certified compiler implementation that have to be trusted. For this reason it is important to make this part as simple as possible. Intermediate language programs are represented within the compiler as abstract datatypes. This kind of representation is easily ported to higher-order theorem provers including Coq. Another important goal is to make the MIPS code formalization look like MIPS assembler code to help people agree that the theorem prover representation generation is indeed a very small gap in the verification chain.

The generation mechanisms for the program representations are implemented in ML in a very simple, standard way. They write each statement or instruction one after another into a theorem prover file and comprise less than 100 lines of code.

3.2 Certificate Generation Algorithm

The certificate generator generates the proof scripts based on the information provided by the compiler and the actual intermediate language and MIPS program code.

3.2.1 Generating the Prerequisites

Since we use a simulation proof we have to generate a simulation relation. It is established using the sets of variables of a program, the variable mapping, and the program counter relation in a way that it demands the correspondence of variables values' to memory and registers and the correspondence of the program pieces from intermediate and MIPS code. Next the certificate generator generates proof scripts

to state and prove the properties of the variable mapping. In our current implementation we where able to simplify this process compared to the earlier Isabelle implementation due to the use of checker predicates (cp. [5]) which encapsulate some operations for which a non-checker predicate implementation whould have required the generation of a more comprehensive proof script.

3.2.2 Generating the Proofs for the Second Phase

The proof scripts for the second phase make use of all the information the compiler omits plus the intermediate language and MIPS program code. For each possible execution step of an intermediate code statement and corresponding piece of MIPS code we make a case distinction on the syntactical structure of the intermediate language statement to be handled. We have different raw proof script templates covering the different cases. Such a template has some blanks in it. The certificate generator fills them up with information from the corresponding MIPS code and from the compiler provided information.

Figure 3 shows a proof script template for an array assignment. It consists of an intermediate language statement it has to be mapped against, an optional MIPS code piece, a priority, and the proof script template that is completed to serve as the generated proof script by filling up names of variables, facts, and case distinctions.

In the figure an array element is assigned the value of another array element plus a second argument. This is shown in the intermediate language part of the pattern (denoted IL) which consists of a term representing possible intermediate language statements. The second argument is presumably a constant since rules have priorities and more complicated cases have more distinct patterns they are matched against and higher priorities. *MemMap* is a function – which is constant to the whole program – realizing the variable mapping. Whereas *varvals* and *mem* are variables representing stores for variables' values and memory from the languages' semantics. Different elements can be distinguished in the proof code:

- The evaluation of terms like in the first two lines.
- The split of a proof goal into sub goals (third line).
- The assertion and proving of some more basic facts (next five lines).
- The rewriting of a term by something equivalent (last line of the first sub-goal).
- The *qed* denotes a sub-goal that can be trivially finished. Usually some kind of equation needs to be proved in the last step.

The parts in this proof template that need to be instantiated are the first four lines in the assertion of the first sub-goal. The corresponding variable names have to be put into locations. Furthermore, if we use user provided facts that prove whether a memory location is a valid index or not some adaptation to the proof script needs to be made. We have implemented a mechanism to reference and manage such facts in the proof script generation. If we do not use user provided facts, we need to make a case distinction on the range of the index. Thus, we need to make some adaptation to the proof script, too. This is done by referencing the index variable's name again. A typical generated proof for such a step consists of roughly about 100 applications of Coq tactics. There are as many step proofs as there are intermediate language

```
IL:
  ASSIGN_AV (a,b,PLUS(ARVAR (c,d),e))
MIPS:
  -
Priority: 10
ProofCode:
evaluate succeeding IL state representation
evaluate succeeding MIPS state representation
split_goal_into_subgoals:
1. memory correspondence:
   assert ( varvals (v, i) = mem (MemMap (v, i))) for all valid v and i
     prove that a is a valid array
     prove that c is a valid array
     prove or use that b is a valid index
     prove or use that d is a valid index
   use injectivity + assertion to rewrite variable accesses with
        memory accesses in variable store representation
   qed
2. program counter correspondence:
   look up whether program counters correspond to each other
   qed
```

Fig. 3. Proof Script Template for an Array Assignment

statements in the intermediate language representation in a compilation run. Note, that in the ML implementation of our certificate generator the proof templates are encoded in a very compact form which outperforms even this pseudo proof code representation in terms of lines of code. We encapsulate common sequences of tactic applications in templates themselves.

3.2.3 Completing the Certificate Generation

The generation of the scripts for the third phase only needs information on the sets of variables, the variable mapping and the program counter relation. The sets of variables and the variable mapping are only needed to prove the correspondence of the initial states.

The generation of the proof script proving the steps correct is by far the most complicated process. The proposed methodology has the advantage that it can be refined during the development process. We can start with a generic proof script for a step of corresponding code pieces that may be able to prove some cases correct. We go on refining this script depending on the statements, expressions, and operators occurring in the intermediate language. We can go further on with the refinement by looking at the compiler provided information and the MIPS code.

The part of our certifying code generation that emits the proof script consists of several hundred lines of ML code.

3.3 Preparing and Using the Theorem Prover

The generated proof scripts and program representations are stored in separate files. These are passed together with correctness criterion and language formalizations as well as pre-proved lemmata and auxiliary formalizations to the theorem prover. The proof scripts are processed one after each other by the theorem prover. The generated proof scripts contain a distinct file stating that the correctness criterion instantiated with intermediate language and MIPS program is fulfilled. This file is processed last, since its correctness proofs depends on all the previous files. With the successful processing of this last file the certificate checking is successfully finished.

4 Results

A comparison between our two implementations [5] shows that the second implementation described here is much faster with respect to the certificate checking: We are now able to verify code generation runs of several hundred lines of intermediate language code within a few minutes. Comparable examples took days to verify with the first implementation. The time for performing the compilation and generating the proof scripts is negligible.

The table shows the time[4] it takes to prove the code generation of different independently compiled procedures correct. It shows the number of variables occurring in the procedure (counting array elements as single variables). The length of the original intermediate program (IL length) as well as the length of the generated MIPS code (TL length). In the last three columns the time it takes to check the proofs for the three different phases (cp. Section 2) is shown.

program	no. variables	IL length	TL length	phase1	phase2	phase3
sort1	1008	16	67	3s	5s	1s
sort1a	2008	16	67	6s	5s	1s
sort1b	3008	16	67	10s	5s	1s
sort1c	4008	16	67	15s	5s	1s
sort1d	5008	16	67	22s	5s	1s
arith1	16	177	705	1s	38s	13s
arith2	18	353	1409	1s	1m 53s	46s
arrays1	2030	520	2059	5s	4m 25s	1m 34s
arrays2	2030	1030	4107	5s	14m 47s	6m 24s

The *sort* procedures sort arrays from 1000 (*sort1*) up to 5000 (*sort1c*) elements. The *arith* procedures mostly contain arithmetic operations while the *arrays* procedures perform operations on differently sized arrays. With procedures reaching several hundred lines of code the time it takes to check the proofs is increasing faster than linear. This is due to the larger data structures which have to be handled during the proof process. Accesses to these structures grow linear with code size however since the structures themselves are growing linear we end up with a time that is growing quadratic. Accesses comprise the look up of statements, instructions, variable, memory and program counter correspondences from list like data structures. These look up operations were carried out in the earlier Isabelle/HOL 2005 based implementation [6] mostly by unfolding the definition of a look up function and matching axioms describing the semantics of such a function against the definition and the data structure containing the data to be looked up. In Coq we are able to execute look up function definitions directly in the Coq environment. Hence, the quadratic time that these look-up operations add to the checking process has only a major effect for very large procedures in Coq. These operations where the bottleneck in our Isabelle implementation. The use of the checker predicate has speeded the first phase up by a factor slightly larger than 100 compared to a Coq implementation without a checker. Due to an artifact in the Coq implementation the time the third phase takes grows larger than linear with the size of the program code. In the presented table we used explicit proof terms for conducting the third

[4] Experiments conducted on Intel Core 2 Duo machine with 2.16 GHz using one core and Coq Version 8.1.

phase. We do have however a version with linear growing time in the third phase which is slightly slower for the last line in the table. A Coq implementation without checkers and explicit proof term use is described in [2].

5 Conclusion and Future Work

Our certifying code generation phase presented in this tool description paper demonstrates that certifying techniques can be used successfully in compilers together with higher order theorem provers as certificate checkers. Despite our good results with the second implementation we are currently working on a third implementation:

- A Coq version of a certifying code generation phase and other compiler phases that relies massively on checkers even in the second phase of our proof scripts.

We expect even more speed gains and a reduction on proof script generation complexity once this version is finished.

References

[1] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *CAV 2005*, volume 3576 of LNCS, pages 291–295. Springer-Verlag, 2005.

[2] J. O. Blech. On certifying code generation. Technical Report 366/07, University of Kaiserslautern, November 2007.

[3] J. O. Blech. Certifying System Translations Using Higher Order Theorem Provers. PhD-Thesis, University of Kaiserslautern, *submitted in April 2008*.

[4] J. O. Blech, L. Gesellensetter, and S. Glesner. Formal verification of dead code elimination in Isabelle/HOL. In *Software Engineering and Formal Methods*, pages 200–209. IEEE, IEEE Computer Society Press, September 2005.

[5] J. O. Blech and B. Grégoire. Certifying code generation with coq. In *Proceedings of the 7th Workshop on Compiler Optimization meets Compiler Verification (COCV 2008), Budapest, Hungary*, ENTCS. April 2008. *to appear*.

[6] J. O. Blech and A. Poetzsch-Heffter. A Certifying Code Generation Phase. In *Proc. COCV Workshop, ETAPS 2007*, ENTCS, March 2007.

[7] B. Buth, K.-H. Buth, M. Fränzle, B. von Karger, Y. Lakhnech, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In *Proc. CC '92*, volume 641 of LNCS, Springer-Verlag, 1992.

[8] M. J. Gawkowski, J. O. Blech, and A. Poetzsch-Heffter. Certifying Compilers based on Formal Translation Contracts. Technical Report 355-06, University of Kaiserslautern, November 2006.

[9] S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender Uebersetzer(Verifix: Construction and Architecture of Verifying Compilers). *it- Information Technology*, 46(5):265–276, 2004.

[10] G. Goos and W. Zimmermann. Verification of compilers. In Bernhard Steffen and Ernst Rüdiger Olderog, editors, *Correct System Design*, volume 1710 of LNCS, Springer-Verlag, November 1999.

[11] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[12] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Software Engineering and Formal Methods*. IEEE, IEEE Computer Society Press, September 2005.

[13] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *proc. POPL'05*, pages 364–377, ACM Press, 2005.

[14] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *proc. POPL'06*, pages 42–54, ACM Press, 2006.

[15] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *proc. PLDI'98*, pages 333–344, ACM Press, 1998.

[16] G. C. Necula. Proof-carrying code. In *proc. POPL'97*, ACM Press, January 1997.

[17] G. C. Necula. *Compiling with Proofs*. PhD thesis, 1998.

[18] G. C. Necula. Translation validation for an optimizing compiler. In *proc. PLDI'00*, pages 83–95, ACM Press, 2000.

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS, Springer-Verlag, 2002.

[20] D. A. Patterson and J. L. Hennessy. *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[21] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *proc. TACAS*, volume 1284 of LNCS, 151+, Springer-Verlag, 1998.

[22] A. Poetzsch-Heffter and M. J. Gawkowski. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):37–51, 2005.

[23] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[24] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1* http://coq.inria.fr.

[25] J.-B. Tristan and X. Leroy. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *POPL '08: Conference record of the 35rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2008. ACM Press.

[26] W. Zimmermann. On the Correctness of Transformations in Compiler Back-Ends. volume 4313 of LNCS, Springer-Verlag, 2006.

[27] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.