

Sebastian Prehn

# Formally certified and certifying compiler back-ends

Term Paper

January 20, 2008

TU Kaiserslautern  
AG Software Engineering  
Seminar WS08



---

## Contents

<b>1</b>	<b>Introduction</b> .....	1
1.1	Can you trust your compiler? .....	1
<b>2</b>	<b>Approaches</b> .....	5
2.1	Correctness Property .....	5
2.2	Certified Compilers .....	6
2.3	Certifying Compilers .....	7
2.4	Proof Carrying Code .....	7
2.5	Translation Validation .....	9
2.6	Summary .....	11
<b>3</b>	<b>Example</b> .....	13
<b>4</b>	<b>Conclusion</b> .....	17
	<b>References</b> .....	19
	Bibliography .....	19



## Introduction

### 1.1 Can you trust your compiler?

Compilers play a key role in software development. Any high level source code has to go through steps of compilation in order to be machine executable. Usually the application developer treats the compiler as a black box. A compiler however is a piece of complex software. And like any complex software it can contain bugs.

Today, compilers are quite well understood and fortunately work quite reliably on standard input. On the other hand, take a look on the long compiler bug list of recent gcc versions<sup>1</sup>. If such an error is present one can only hope to discovered such a bug during testing of the application. However one can not rely on a correct translation into executable code.

It turns out that defining the term "correct translation" is not that easy<sup>2</sup>. What would be the correct translation of the following code?

```
#include <stdio.h>
int main() {
    int i;
    int array[] = {0,0,0};
    i = 1;
    array[i] = ++i; // evaluation order ?
    printf("[ %d, %d, %d]\n",array[0], array[1], array[2]);
    return 0;
}
```

On gcc (GCC) 4.1.2 (Gentoo 4.1.2) the result is "[ 0, 2, 0]". However this outcome is not guaranteed. It may vary due to undefined behaviour of when the i variable is evaluated, before or after the increment operation (++). An

---

<sup>1</sup> GCC Bug Database <http://gcc.gnu.org/bugzilla/>

<sup>2</sup> You will find a more detailed discussion on the correctness criteria in chapter 2.1.

outcome of “[0, 0, 2]” would be also be legitimate - also correct. This deviation is considered a non-bug, since the behaviour is underspecified.

Similarly a lot of uncertainty is introduced when conduction floating point computation.

```
#include <stdio.h>

int main() {
    double a = 0.5;
    double b = 0.01;
    printf("result: %d\n", (int)(a / b)); // cast double to int
    return 0;
}
```

This little program might print “result: 50” on some systems and optimization levels, and “result: 49” on others. This is not a bug in the compiler, but an inherent limitation of the floating point types [1].

Obviously, what is considered a correct translation depends on the defined semantics of a language. Nevertheless let us turn to a simple and unambiguous example. Consider this simple C program:

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    int result, i;
    i = 0;
    result = -2 * abs(i-2); // error
    printf("result: %d\n", result);
    return 0;
}
```

The expected result of the computation would be  $-2*|0-2| = -2*2 = -4$ . The result actually returned by the compiled binary is 4. This is a known compiler bug (bug number 34130, see [1]) in GCC (3.2 and 4.1) reported 2007-11-17 01:21 and still present at the time of writing. In this case it turns out there is an error in the compiler assuming  $a * |b|$  to be equal to  $|a * b|$ .

This bug could have disastrous consequences and it is very unlikely that an application developer would have written a test case for correct absolute value calculation. Just by chance could this bug be discovered.

So what are the benefits of trusted compilation? Let us assume the developer had noticed that the program execution is incorrect. He would most likely blame his own source code first. It could take him days to identify the problem as a compiler error. In trusted compilation the compiler bug would have been discovered. Trusted compilation can therefore positively impact the development process [2].

Production ready compilers must not contain bugs and therefore have to be well tested. Usually stable/old versions and low optimization levels are assumed to provide these qualities. However, modern languages impose new requirements on compilers. The challenges include code generation for declarative, distributed or parallel computing. There is a trend towards more readable and easy to use code. The optimization is left to the compiler. Thus it is necessary to constantly enhance a compiler. However, the need for a stable compiler freezes compiler development. It takes too long for new scientific findings to make it into the stable version [2]. Recent development in trusted compilation targets the ease of compiler evolution while assuring correctness properties on the translation (see chapters 2.3 & 2.5).

Critical, high-assurance software requires the use of formal methods (model checking, program proof, etc). These methods are applied to the the source code in order to guarantee safety properties. However, an uncertified compiler can invalidate all these guarantees making all prior verification steps useless. In security relevant applications, the compiler is actually the weak link between certified source code and certified hardware. [3]

This rest of this work is organized into the following two chapters:

- Chapter 2, the major part of this thesis, gives an overview of the possible solutions to the compiler verification problem, without going into detail of the formal verification techniques.
- Chapter 3 shows an example of how a single compiler transformation can be semi-formally verified.
- Chapter 4 concludes the work.



## Approaches

For the following considerations let there be some agreements on the notation.  $S$  designates the set of sources.  $C$  stands for the set of compiled code. In cases where the compiler does not succeed in producing compiled code the failure is denoted by an error *None*, otherwise the result is *Some(c)*<sup>1</sup>. Thus, a compiler is a total function *Comp* from the set of sources  $S$  to either  $C$  or  $\{None\}$ .

$$Comp : S \rightarrow C \cup \{None\} \tag{2.1}$$

### 2.1 Correctness Property

When is a program translation correct? The process of translation is supposed to be transparent, meaning the semantics of the source code is preserved. To put it more concrete the notion of simulation has been established. This means the compilation has to simulate the source in all observable behaviour. In this case source and compilation are in a simulation relation. Formally this relation is established once for every behaviour in the source language there exists an equivalent behaviour in the target and vice versa and these matching pairs are mapped by the compiler function.

In general the certification of a compiler guarantees that all safety properties proved on the source code shall hold for the compiled code as well [3]. Often chosen properties are type and memory correctness. Please note that these two property instances alone are too weak to ensure any simulation relation between source and compiled code.

Xavier Leroy [3] identifies five possible correctness properties  $Prop(s,c)$  between source  $s \in S$  and compilation  $c \in C$ .

1. “ $s$  and  $c$  are observationally equivalent.”
2. “If  $s$  has well-defined semantics, then  $s$  and  $c$  are observationally equivalent.”

---

<sup>1</sup> Commonly used in functional programming languages: see data type “option”

3. “If  $s$  has well-defined semantics and satisfies the functional specification  $Spec$ , then  $c$  satisfies  $Spec$ .”
4. “If  $s$  is type- and memory-safe, then so is  $c$ .”
5. “ $c$  is type- and memory-safe.”

Full observational equivalence (property 1) also requires the compiled code to be observationally equivalent even if the semantics of the source are not well-defined. Compilers are usually free to choose a behaviour in such undefined situations. Therefore property 1 is too strong and undesired.

For every specification  $Spec$  specifying observable behaviour only, which is generally the case, property 2 implies property 3. Property 4 is an instance of property 3. This is why property 2 is important in practise. It covers many individual cases. Property 5 is a typical case where the source code doesn't play a role. The compiler ensures that the compiled code is type- and memory-safe.

The selected property depends on the field of application, as we will see in the following chapters. In some fields a certificate on type- and memory-safety may be sufficient, where as in other fields the correctness of the transformation must be ensured. Why do we not always require observational equivalence? Property 2 can be much harder to prove than let's say property 4.

## 2.2 Certified Compilers

A certified compiler is a compiler, whose algorithm and implementation are verified. It ensures defined safety properties and is therefore suited to compiler critical software systems.

$$\forall s \in S, c \in C : Comp(s) = Some(c) \Rightarrow Prop(s, c) \quad (2.2)$$

In order to provide a consistent chain of proof every single transformation requires a proof. Furthermore it has to be distinguish between a proof on an algorithm and proof of its correct implementation. Both have to be present. These proofs can be quite difficult and complex and even proving simple thesis can be very labor-intensive, if done by hand. In [4] it is shown how the process of proving optimization correctness of the implementation can be automated.

Assumed all proofs are complete and correct this approach ensures having no bugs in the compiler. It has several advantages over other certifying approaches (see chap. 2.3): Compilation without the additional overhead due to self-checking mechanisms at compile time is more efficient. A program with correct semantics will always compile.

However, no matter if the process of proving can be automated, putting up the proof obligations for each transformation is left to be done manually.

In conclusion the process of developing a certified compiler is still too heavy and freezes compiler development. Consequently, there are no production-ready, freely available certified compilers on the market<sup>2</sup>.

## 2.3 Certifying Compilers

Certifying compiler approaches address the issue of complicated compiler development in trusted compilation. In general it is easier to prove a correct computation than to prove the correctness of the computation. Certifying Compilers take advantage of this fact, in that they equip a compiler with a certifier that checks the result of a translation.

A certifying compiler  $CComp$  is a compiler function that either fails ( $CComp(S) = None$ ) or returns both a compiled Code  $c \in C$  and a proof  $\pi \in \Pi$  for the property  $Prop(s,c)$ .

$$CComp : S \rightarrow (C \times \Pi) \cup \{None\} \quad (2.3)$$

There is no guarantee that the returned proof  $\pi$  will work. But it can be checked in a second step. This can be automated by a theorem prover. If the proof works, the compiled code satisfies  $Prop(s,c)$ .

$$\forall s \in S, c \in C, \pi \in \Pi : CComp(s) = Some(c, \pi) \wedge \pi correct \Rightarrow Prop(s, c) \quad (2.4)$$

## 2.4 Proof Carrying Code

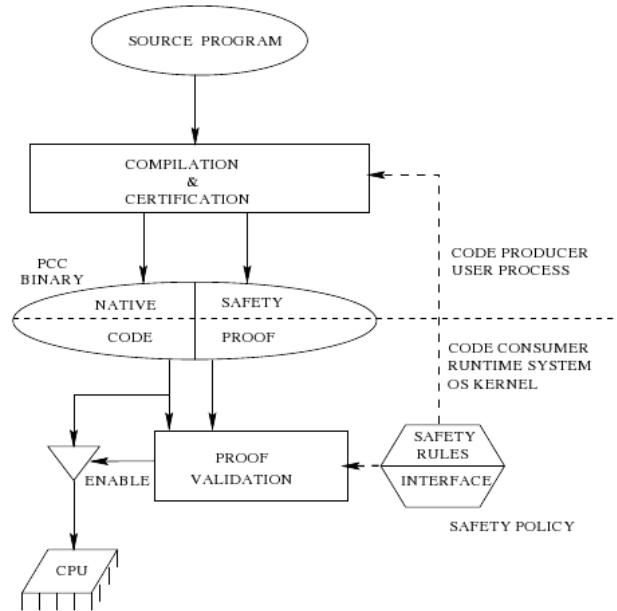
Proof Carrying Code [5] is an approach originally developed for safe execution of untrusted code. It applies e.g. to a mobile setting (e.g. browser applets, software for cellular phones or PDAs) where code user and code producer differ and are distributed over a network. The traditional approach in such a scenario is to select a trustworthy code source<sup>3</sup> and transport the code to the client by means of a secure communication channel<sup>4</sup>. Hence, the code provider is held credible for the code quality.

The idea of Proof Carrying Code is to eliminate the need for trust between code consumer and code provider, as well as the need for secure communication. Instead the code itself carries a proof on predefined safety properties. The proof is generated by the code producer during compilation (see figure 2.1). Proof Carrying Code does not make a statement on who generates the proof.

<sup>2</sup> to my best knowledge

<sup>3</sup> often done via certificates

<sup>4</sup> such as https and / or using check-sums



**Fig. 2.1.** Overview of Proof Carrying Code [5]

It can be generated by a compiler, but it could also be generated by hand. Subsequently, the client validates the proof to verify the safety properties.

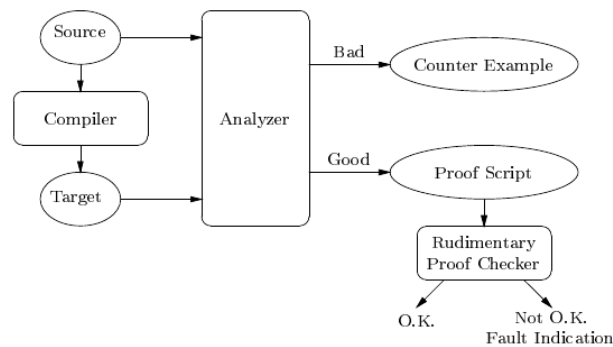
In case of malicious code the proof fails and the code will not be executed. If the proof succeeds the client can safely execute the code. In this approach there is not even the need for a secure communication, since it does not matter if proof or code was manipulated. If the manipulation broke the proof then the program would not be executed. If the manipulation did not break the proof then the execution is still safe. This technique comes with a small overhead on performance: Prior to the first execution the proof has to be checked. Since checking a proof for correctness is much simpler than coming up with the proof in the first place, this validation step can be done quickly. Once the code is verified on the code consumer the code runs without performance drawbacks.

One example on how Proof Carrying Code could be utilized is given by [6]. Proof Carrying Code is suggested to check Java applications on mobile devices. The safety properties of interest in such a mobile scenario are usually Type- and Memory-Safety. This corresponds to property 5 (see chapter 2.1). Thus, Proof Carrying Code can as well be applied to trusted compilation. In case of the proof being generated by the compiler one can speak of a certifying compiler. Please note that this approach works only for properties that can be checked on the compilation. For the compilation process this means that

the Proof Carrying Code technique can be used to detect bugs in a compiler. However it can not guarantee a correct translation or a simulation relation between source and compilation.

The overall advantage of applying Proof Carrying Code to trusted compilation is: No formal verification of compiler itself is needed, since there is no need to trust the code producer, which is the compiler. Instead one has to rely on the correctness of the proof checker (i.e. a theorem prover).

## 2.5 Translation Validation



**Fig. 2.2.** Overview of Translation Validation [7]

Certification that compilation correctly simulates the source (property 2) is desired. Proof Carrying Code is not suitable for such a task, as seen in chapter 2.4. Proving the correctness of a compiler is very complicated and freezes development, as seen in chapter 2.2. George C. Necula [8] observed: “If we can not prove that a compiler is always correct maybe we can at least check the correctness of each compilation.” This describes very well the idea of Translation Validation. In Translation Validation each compilation result is automatically checked against the source and thus compilation errors can be detected on-the-fly.

Translation Validation was first presented in [7]. It is a very general approach that can be used to verify any kind of translator. The translator may be a complete compiler, one single compiler pass or another form of code generator. In this approach a verifier monitors each transformation and ensures that the transformed code still simulates the source code. If it can not verify this relation the transformation is aborted. In consequence the result of a successful transformation must be a correct simulation of the source program, i.e. a certified compilation.

For Translation Validation to work there has to be a clear foundation of what a correct simulation is. Note that in literature simulation, refinement, and implementation are used synonymously, in this case. The formalization of this notion is based on a mathematical relation between source and compilation.

Therefore a common semantic framework which correlates source and a corresponding representation in the target language must exist<sup>5</sup>. This way the whole process of translation validation can be automated to find corresponding sections in source and target and automatically prove the simulation relation.

In order to avoid compiler freezing, as described in chapter 1.1, Translation Validation leaves the compiler itself untouched. How the output is generated is unknown to the verifier.

In [7] an architecture for a Translation Validation framework is proposed. The verifier is split into two components, an analyser and a proof checker ( see figure 2.2). Both source and target are fed into analyser. The analyser finds corresponding sections in the code and generates proof obligations in form of proof scripts for the checker. The proof checker ( a simple theorem prover) then has to verify the correctness of the proof scripts. Either analyser or proof checker may fail. However a correct Translation Validation Verifier must ensure that the following equation holds<sup>6</sup>.

$$\forall s \in S, c \in C : \text{Comp}(s) = \text{Some}(c) \quad \wedge \quad \text{Verify}(s, c) = \text{true} \quad (2.5) \\ \Rightarrow \text{Prop}(s, c)$$

This does not mean that every correct program must verify, but every verified program must be correct (in terms of Prop(s,c)). A Translation Validation compiler that returns *None* for all input is correct. Of course, the goal is to provide a compiler that generates target code at a low to zero rate of false-negatives. This is a quality of implementation issue.

The downside of Translation Validation is that correct translations may be rejected. On the other hand, the unintrusiveness of this approach may lead to quicker compiler development cycles, since the verifier is certified once, but may change frequently and must not be verified. According to [8], the overhead for writing the verifier is supposed to be comparable to the effort of implementing one compiler pass.

In some cases it can be useful to combine both approaches, Proof Carrying Code and Translation Validation, in that sense that the compiler is modified to produce additional code annotations to ease the task of the Translation Validation analyser [3]. For example additional variable type annotations can help the verifier to check type safety. So equation 2.6 changes for Translation Validation with Annotations *A* to:

---

<sup>5</sup> Relate semantic of each statement in source to target. Example: variable assignment in source and variable assignment in target

<sup>6</sup> In consequence the verifier itself must be formally verified.

$$\begin{aligned} \forall s \in S, c \in C, a \in A : \text{Comp}(s) = \text{Some}(c, a) \quad \wedge \quad \text{Verify}(s, c, a) = \text{true} \\ \Rightarrow \text{Prop}(s, c) \end{aligned} \tag{2.6}$$

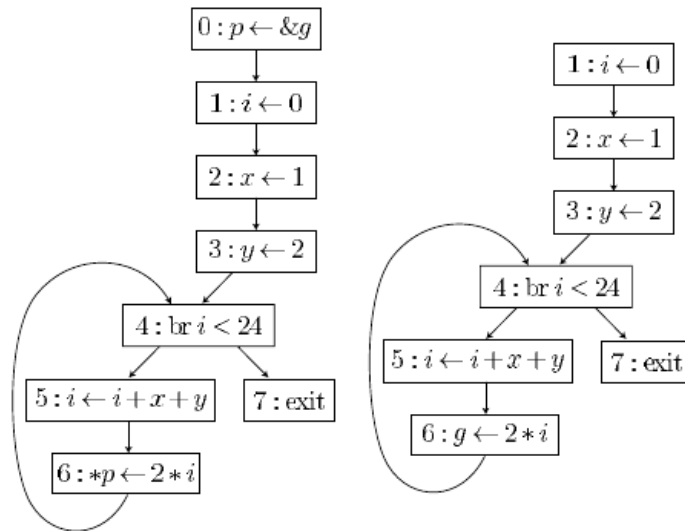
## 2.6 Summary

Compilers are too complicated to certify. This is why formally certified compilers have not prevailed and certifying compiler approaches have emerged in the first place. Proof Carrying code is a technique that was borrowed from another field of science, mobile computing. It is only partly suitable for trusted compilation, mainly because it is not powerful enough to prove the simulation relation. Translation Validation is the latest and most promising approach. Its only drawback is, that a correct translation might be rejected due to deficits in the verifier implementation. However, in theory it is powerful enough to prove the simulation of observable behaviour and will ease compiler development in the future.



## Example

Proving the simulation relation is no simple task. Especially when it must be done automatically. The following example shows how such a proof can be constructed, for a special case: pointer elimination. It is an example<sup>1</sup> of what needs to be done in a Translation Validation verifier.



**Fig. 3.1.** Original Program (left), Program After Pointer Elimination (right) [2]

Figure 3.1 illustrates two control flow graphs of a program in an intermediate language before and after pointer elimination. The program excerpt

<sup>1</sup> Example taken from [2]

contains several variable assignment nodes, a conditional branch<sup>2</sup> (label 4) and a program exit statement (label 7). During a translation phase the compiler has eliminated the assignment for variable  $p$  to the address of  $g$  (label 0) and substituted  $g$  for every occurrence of  $*p$  (label 6). The verifier must now prove that the transformed program simulates the original program in all observable behaviour. The verifier does not prove that the compiler always performs correctly, but that this particular transformation was correct.

In order to prove this thesis the verifier could be programed to prove the correctness in two steps<sup>3</sup>:

1. Analysis is correct:  $p$  always points to  $g$  when node 6 executes (denoted by pre-condition invariant:  $\langle p = \&g \rangle 6$ ).
2. Transformation is correct: use 1. to prove that both results are identical.

In order to prove the analysis correct, the compiler can try to prove that  $p = \&g$  holds between assignment of  $p$  and just before every execution of node 6<sup>4</sup>.

This proof can be done by structural induction over the partial execution of the program. The basis should be clear: right after the assignment at node 0  $p = \&g$  is true, so  $\langle p = \&g \rangle 1$  is satisfied. To prove  $\langle p = \&g \rangle n$  for label  $n$  let there be  $M$  the set of all directly preceding node labels of node  $n$ . The induction thesis assumed is:  $\forall m \in M : \langle p = \&g \rangle m$ . The induction step will succeed for all nodes, since no assignment except the one in 0 affects variable  $p$ <sup>5</sup>. This induction can be automated by supplying an appropriate calculus (see section 5.3 in [2]).

In order to prove the the transformation correct, the compiler has to ensure that all input/output relations and all global variables are simulated correctly. In this example variable  $g$  shall be the only global variable. Variable  $g$ 's final value must be equivalent in both program versions.

In order distinguish variables and labels from the original program and the transformed program, elements from the original program carry the subscript "P" and elements from the transformed program carry a "T". The program exits at label 7. So the key simulation invariant is  $g_P$  at  $7_P$  equals  $g_T$  at  $7_T$ , denoted by pre-condition invariant  $\langle g_P \rangle 7_P \triangleleft \langle g_T \rangle 7_T$ . Let us assume that the verifier determines corresponding program states simply by matching node labels<sup>6</sup>. The proof is by induction over the partial execution of

<sup>2</sup> On evaluation "true" the left branch is taken, on "false" the right branch is taken

<sup>3</sup> This is knowledge, the verifier developer puts into the verification stage.

<sup>4</sup> This strategy might not succeed in every case where  $\langle p = \&g \rangle 6$  holds. It is a quality of implementation issue of the verifier. Giving a false negative is still a correct behaviour (see chapter 2.5)

<sup>5</sup> To prove  $\langle p = \&g \rangle 4$  it is important that  $\langle p = \&g \rangle 6$  is the induction thesis and therefore  $p \neq \&p$  holds.

<sup>6</sup> Success of this strategy depends on the kind of optimization performed. For a loop unrolling compiler pass this strategy would not be sufficient.

the transformed program. In order for this proof to succeed the invariants at the nodes between 1 and 7 have to be enriched to reflect the correspondence of the intermediate steps. Those are the invariants generated:

$$\langle g_P \rangle 1_P \triangleleft \langle g_T \rangle 1_T \quad (3.1)$$

$$\langle (g_P, i_P) \rangle 2_P \triangleleft \langle (g_T, i_T) \rangle 2_T \quad (3.2)$$

$$\langle (g_P, i_P, x_P) \rangle 3_P \triangleleft \langle (g_T, i_T, x_T) \rangle 3_T \quad (3.3)$$

$$\langle (g_P, i_P, x_P, y_P) \rangle 4_P \triangleleft \langle (g_T, i_T, x_T, y_T) \rangle 4_T \quad (3.4)$$

$$\langle (g_P, i_P, x_P, y_P) \rangle 5_P \triangleleft \langle (g_T, i_T, x_T, y_T) \rangle 5_T \quad (3.5)$$

$$\langle (g_P, i_P, x_P, y_P) \rangle 6_P \triangleleft \langle (g_T, i_T, x_T, y_T) \rangle 6_T \quad (3.6)$$

$$\langle g_P \rangle 7_P \triangleleft \langle g_T \rangle 7_T \quad (3.7)$$

Invariant 3.1 comes for free. The global variable  $g_P$  and  $g_T$  are assumed to be equal prior to program executions. This serves as the induction basis.

During the execution of the next two nodes (1,2) in both programs variables  $i$  and  $x$  are set to the same values. The invariants are enriched. None of the other variables in the preceding invariant are affected, however. So invariants 3.2 and 3.3 hold.

Invariant 3.4 is a little more tricky. In the original program there are two execution paths leading to node  $4_P$ . Node  $4_P$  can be reached through node  $3_P$ . In this case the same arguments as in the last section apply. The same arguments applies for node  $4_T$  being reached through node  $3_T$ . Node  $4_P$  can also be reached through node  $6_P$  ( $4_T$  and  $6_T$  respectively). Assuming that invariant 3.6 holds, 3.4 follows, since from the analysis phase the verifier can be sure that  $*p = g_P$  at  $6_P$ .

Trivially, invariant 3.5 holds, since execution of neither program alter an invariant variable in the *while* statement.

Invariant 3.6 holds. This is because both programs calculate the new value for  $i_P$  and  $i_T$  from as the sum of equivalent variables. The equivalence is ensured by invariant 3.5.

Finally, 3.7 holds since the only predecessor is node 4. Node 4's simulation invariant holds. So the while statements in both programs will agree on the same branch. For the same reason must  $g_T$  and  $g_P$  be equal, since neither program alters the  $g$  variable in the *while* statement.

This induction can as well be automated by supplying an appropriate calculus (see section 5.4 in [2]).



## Conclusion

Nobody can deny the benefit of having a certified compiler. Certified compilation is and will become more and more an important topic, since it is a mandatory techniques to facilitate development of new compiler optimizations and features.

There has been research for over a decade now, so one might ask: Are there any production ready certified compilers available? Unfortunately there are still no such compiler available<sup>1</sup>. In the year 2000, George C. Necula has shown that it is possible to equip the GCC GNU-C compiler with a Translation validation verifier (see [8]). Xavier Leroy has presented a C-minor compiler<sup>2</sup> in 2006 (see [3]) for a subset of the C programming language. This shows that there are efforts made into the right direction. However these implementation seem not to be production ready. Even if they were, it should be clear that the C programming language is probably the closed language to assembly code. More modern, higher-level languages like C# or Java and the ever increasing requirements<sup>3</sup> on the compiler software impose much more complicated challenges.

Writing certifiers for such complicated tasks will be quite tough. As a benefit, however, the compiler development process will be enabled to incorporate current research results much quicker.

---

<sup>1</sup> at least, to my best knowledge

<sup>2</sup> using the Coq proof assistant

<sup>3</sup> optimization for concurrency etc.



---

## References

1. *GCC Bugs - GNU Project - Free Software Foundation (FSF)*. <http://www.gnu.org/software/gcc/bugs.html> November 2007.
2. M. Rinard und D. Marinov. *Credible Compilation with Pointers*. (July 1999).
3. Xavier Leroy. *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*. (2006).
4. Sorin Lerner, Todd Millstein und Craig Chambers. *Automatically Proving the Correctness of Compiler Optimizations*. (November 2003).
5. George C. Necula. *Proof-Carrying Code*. In *In Proceedings of the 1997 IEEE Symposium on Security and Privacy* 1997.
6. Michael Franz, Deepak Chandra, Andreas Gal, Vivek Haldar, Fermn Reig und Ning Wang. *A Portable Virtual Machine Target For Proof-Carrying Code*. (Jun 2003).
7. Amir Pnueli, Michael Siegel und Eli Singerman. *Translation Validation*. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems* pages 151–166 London, UK 1998. Springer-Verlag.
8. George C. Necula. *Translation validation for an optimizing compiler*. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* pages 83–94 New York, NY, USA 2000. ACM.