COCV'04 Preliminary Version

Towards Proof Generating Compilers

Arnd Poetzsch-Heffter [a,1] Marek Gawkowski [a,2]

[a] *Computer Science Department*
University of Kaiserlautern
Germany

Abstract

Correctness of compilation is important for the reliability of software. New techniques to guarantee correctness do not verify the compiler itself, but check for each compiled program whether it is correctly translated. Following these ideas, we developed an approach in which checking is realized as proof checking within a formal specification and verification framework. Based on formal specifications of source and target language and a translation predicate, compilers produce, in addition to the target program `c`, a proof that `c` is correct w.r.t. its source program. This proof can be checked independently of the compiler by the framework. Thus, it can be used as a translation certificate.

The paper describes the overall approach and applies it to a simple translation scenario. Specification and verification is done within the theorem prover Isabelle/HOL. To show the flexibility of the approach, we present two different proof techniques for translation correctness.

Key words: compilation correctness, translation validation, formal translation contract, automatic proof generation, proof checking

1 Introduction

Most software systems are implemented in high-level programming languages. Thus, they rely on a correct translation to machine code. Correct translation is indispensable for dependable systems. For other kinds of software development, it saves the cost of reporting, finding, and eliminating application errors resulting from incorrect compilation. Although research on compilation correctness is a rather old area, today's production compilers are not verified and do not give guarantees about the correctness of their work. Three major problems have to be solved for a practical realization of correct translation:

[1] Email: `poetzsch@informatik.uni-kl.de`
[2] Email: `gawkowsk@informatik.uni-kl.de`

(i) Definition of correct translation: Assuming a formal semantics of source and target language, it seems to be simple to formalize a notion of a correct translation. However, looking closer at the problem, it turns out that language specifications often neglect some aspects that are important for the definition of correct translation (see [2]):
  - The relation between the data types of source and target language.
  - Observability of states, that is, a specification which states of the source program are considered observable from outside of an execution.
  - Treatment of resource restrictions.

(ii) Complexity: Modern programming languages and translation techniques have become increasingly complex. The quick progress in these areas have made it difficult for compilation correctness to follow.

(iii) Compiler implementation correctness: For compilation correctness, it is not sufficient to prove that the high-level program that describes the compiler is correct. One has to show as well that this program is translated correctly and runs on a correct system.

To avoid the last problem, many researchers in the area have concentrated on *translation correctness* in the last years. The basic difference between compiler correctness and translation correctness is as follows: instead of proving that the compiler implementation correctly translates all admissible source programs, the compiler is enhanced by techniques that provide evidence for the correctness of the translations each time the compiler is applied to a source program. The approaches to translation correctness differ in what form the evidence takes and how it is generated.

Proof Generating Compilers

In [8], M. Rinard presents an approach to translation correctness in which the compiler generates a proof that the target program correctly implements the source program. He calls this approach *credible compilation*. We build on this idea and extend it towards an independent and more flexible proof framework. Figure 1 shows the overall picture. The *proof generating compiler* takes a source program `S` and produces (a) the target program `T` and (b) a script of the proof that `T` is a correct translation of `S`. The proof is done in a formal specification and verification framework, *SVF* for short, that:

- contains specifications of the source and target language,
- contains the specification of a predicate `ctrans(S,T)` expressing that a target program `T` is a correct translation of a source program `S`,
- allows addition of further specification parts and to derive properties from the specifications for use in the generated proofs,
- supports formal proofs and their representations by proof scripts,
- provides a proof checker for formal proofs; in particular the proof checker can check whether a proof script proves `ctrans(S,T)`.

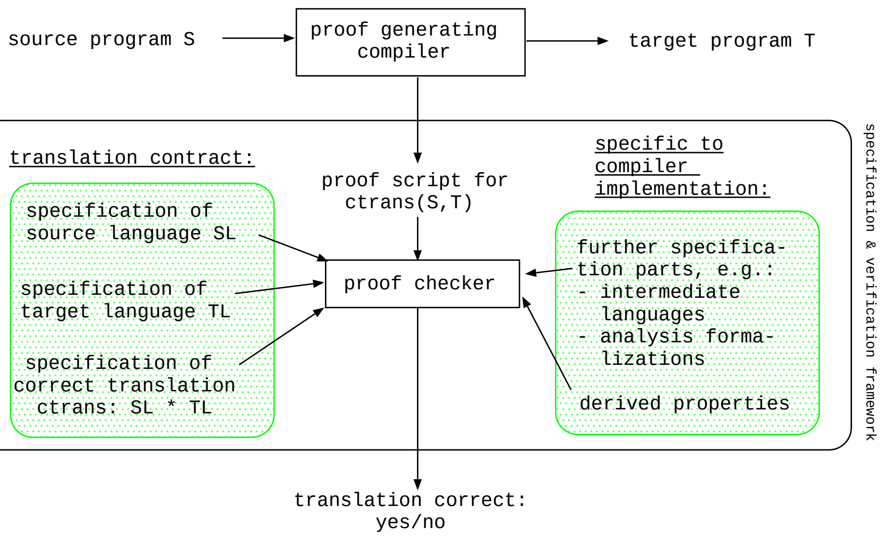
source program S
proof generating compiler
target program T
translation contract:
specification of source language SL
specification of target language TL
specification of correct translation ctrans: SL * TL
proof script for ctrans(S,T)
proof checker
specific to compiler implementation:
further specification parts, e.g.:
- intermediate languages
- analysis formalizations
derived properties
specification & verification framework
translation correct: yes/no

Fig. 1. Proof generating compilation and checking

Our approach to translation correctness is characterized by (a) a clear separation between the two language specifications, (b) an explicit specification of translation correctness, (c) the flexibility gained by supporting further specifications and deriving program independent properties from the given specifications, and (d) an explicit translation certificate in the form of a proof script. It provides flexibility in different directions. Language specifications can be taken as they are and adapted to the special needs of translation correctness within the SVF without loosing correctness. This helps to reuse language specifications and simplifies the definition of when a translation is correct (problem 1 above). In addition, the approach allows for different proof techniques and supports the introduction of intermediate languages (as further specifications) without affecting the original proof goal (this can help to master complexity; problem 2 above). As said above, the problem of implementation correctness is avoided by using translation correctness.

In our approach, the SVF is a general tool that is typically provided by a third party and not by the compiler developer. The SVF has to be sufficiently powerful to specify and reason about programming languages. It should guarantee that specifications are consistent by construction. Ideally, the SVF should provide a powerful prover and a proof checker with a simple to verify implementation and an open architecture. This would allow compiler users with critical applications to inspect the proof checker or ask trusted third parties to do so. Currently, most available SV frameworks do not support a separate proof checker. But, we assume that with the increasing importance of proof carrying code and checking technology, this situation will change soon. For the techniques developed in this paper, we used Isabelle/HOL ([6]) as SVF.

The key point of our approach is that compiler users and compiler writers only have to agree on the *translation contract*, that is, on the language specifications and the specification of translation correctness. These parts are stated explicitly and are independent of the compiler implementation. The compiler writer can introduce further specification parts that usually depend on the internal structure and techniques of the compiler (see Fig. 1). However, the certifying proofs have to show that the translation contract is fulfilled. Altogether, a generated proof script provides checkable evidence that a translation is correct.

Related Work

Although this work has profited from several publications on compiler correctness in general (e.g. [2,9]), we restrict the discussion here to the more closely related work on translation correctness. As already said above, our approach shares the basic idea with the credible compilation approach as presented in [7,8]. The approaches differ in how specification and verification is done. In the papers of Rinard and Marinov, the main contributions are logical rules for verifying transformations on flow graphs. These rules are manually proven sound with respect to an operational semantics of the flow graphs. They develop elaborate techniques for several kinds of optimizations based on powerful program analyses. Our focus is different. We use an existing SVF and argue to separate the language specifications from aspects depending on the compiler implementation.

In our approach, the compiler has to construct the correctness proof. Thus, we assume full instrumentation of the compiler. Approaches to so-called *translation validation* try to avoid this and focus on automating the validation process with little or no change to the compiler. The basic idea is to construct a validation tool which, after every run of the compiler, formally confirms the correctness of the translation. In [10], the background of a validation tool VOC-64 for optimizations of the SGI Pro-64 compiler is described. Similar to the work of Rinard, it works on flow graph representations. VOC-64 can handle structure preserving and structure modifying optimizations. It annotates the given flow graph by invariants and automatically generates verification conditions. Necula [4] describes his work on a translation validation infrastructure for a C compiler which is based on specific rules for evaluation checking.

An approach even closer to compiler correctness proofs is described in [2]: Instead of proving the usually complex translation algorithm correct, it adds a result checking procedure to the compiler that is run after translation. A compilation is only accepted if the checking procedure gives its ok. This approach simplifies compiler verification, because it suffices to verify that the checking procedure is correct for all inputs. The approach still faces the problem of implementation correctness (see above). If we consider this kind of checking approach, translation validation, Rinard's approach with a specific logic, and our approach based on a given SVF, then we can observe a stepwise increase w.r.t. the degree of formalization and separation of checking and compilation.

In [3], Necula and Lee described certifying compilers. A certifying compiler produces for each source program `S` a certificate that the corresponding target program `T` has a certain property. A typical property of interest is type safety. Note that certifying compilers guarantee a property only depending on `T` whereas translation correctness is a property depending on `S` and `T`. However, what is related to our work, is the clear separation between the compilation infrastructure and the checkable ceritificate. That is why many techniques developed for proof carrying code apply as well to our approach (for example, work on the coding of logical frameworks, see [1]).

Overview
To explain the important aspects of our approach, we present its application to a tiny translation scenario, namely the translation of a simple assignment language to a stack-machine language. In Section 2, we formally specify the translation contract. Sections 3 and 4 explain two different proof techniques to illustrate the overall approach and its flexibility. The first technique is directly built on the semantics-based definition of correct translation. The second technique uses an intermediate specification layer that states that correct compilation can be derived if certain syntactical relations between source and target program hold.

2 Translation Contract

A translation contract consists of two language specifications and a predicate stating that a program `T` is a correct translation of a program `S`. In general, the *translation predicate* is necessary

- to relate the input and output types of the different languages,
- to identify observable states of the computations,
- to relate the computation states of the languages.

To keep things simple here, we only describe translation from abstract syntax to abstract syntax. The incorporation of techniques for handling program representation in concrete syntax is considered further work.

The following three subsections introduce the Isabelle/HOL formalizations of two toy languages, $\mathcal{S}$ and $\mathcal{T}$, and of a corresponding translation predicate `ctrans`. Thus, `ctrans S T` expresses the fact that `T`, a program in target language $\mathcal{T}$, is a correct translation of `S`, a program in source language $\mathcal{S}$, as defined by `ctrans`.

2.1 Formalization of the $\mathcal{S}$ Language

The programs `S` of the source language $\mathcal{S}$ are sequences of assignments followed by an expression that yields the result of `S`. Here is a simple `S` program:

```
v0 := 7 + v0; v1 := 8; v0 + v1;
```

Notice that we allow variables to be used before definition. Such variables are considered to be the input parameters of the program. In Isabelle/HOL, the abstract syntax of $\mathcal{S}$ is specified as follows:

$$
\begin{array}{llcl}
\textbf{typedecl} & \textit{variable} & & \\
\textbf{datatype} & \textit{expression} & = & \texttt{INT}\ \textit{int} \mid \texttt{VAR}\ \textit{variable} \mid \textit{expression}\ \oplus\ \textit{expression} \\
\textbf{datatype} & \textit{assignment} & = & \textit{variable}\ :=\ \textit{expression} \\
\textbf{types} & \textit{Sprogram} & = & (\textit{assignment list})\ \times\ \textit{expression}
\end{array}
$$

The type *variable* is abstract and represents the set of variables in a program. There will be a concrete datatype *variable_* for each program. This technique is used to simplify proofs.

Computation states of $\mathcal{S}$ programs are partial mappings from variables to integers. The evaluation of expressions and the computation of assignments is recursively defined. Evaluation is made total by yielding an arbitrary value if a state `s` is undefined for a variable `v`, i.e. if `s(v)` = `None`. This is done using the function `the` of the type α *option* $\Rightarrow \alpha$ which has the following definition: λ`v . case v of Some x` $\Rightarrow$ `x` $\mid$ `None` $\Rightarrow$ `arbitrary`. See Appendix for the description of Isabelle/HOL formalisms we used here.

$$
\begin{array}{l}
\textbf{types}\ \textit{state} = \textit{variable} \rightsquigarrow \textit{int} \\
\texttt{eval} :: \textit{expression} \Rightarrow \textit{state} \Rightarrow \textit{int} \\
\texttt{eval}\ (\texttt{INT i}) \quad \texttt{s} = \texttt{i} \\
\texttt{eval}\ (\texttt{VAR v}) \quad \texttt{s} = \texttt{the (s v)} \\
\texttt{eval}\ (\texttt{e}_1 \oplus \texttt{e}_2)\ \texttt{s} = (\texttt{eval e}_1\ \texttt{s}) + (\texttt{eval e}_2\ \texttt{s}) \\
\texttt{comp} :: \textit{state} \times (\textit{assignment list}) \Rightarrow \textit{state} \times (\textit{assignment list}) \\
\texttt{comp}\ (\texttt{s}, []) \qquad = (\texttt{s}, []) \\
\texttt{comp}\ (\texttt{s}, (\texttt{v} ::= \texttt{e})\#\texttt{as}) = \texttt{comp}\ (\texttt{s}(\texttt{v} \mapsto (\texttt{eval e s})),\ \texttt{as}) \\
\texttt{Comp} :: \textit{state} \Rightarrow \textit{Sprogram} \Rightarrow \textit{int} \\
\texttt{Comp s (as, e)} \equiv \texttt{eval e (fst(comp (s, as)))}
\end{array}
$$

A state `st` is called *appropriate* for a program `S`, if `st` is defined for all variables of `S`. The set of appropriate states of a program is defined as follows:

$$\texttt{appr S} \equiv \{\texttt{st . (Vars S) = (dom st)}\},$$

where the auxiliary function `Vars` :: *Sprogram* $\Rightarrow$ *variable set* yields the program variables occuring in a source program. Appropriateness is later used (a) to express which initial states are acceptable for a program and (b) to prove that all variables of a program have defined values during computation.

2.2 Formalization of the $\mathcal{T}$ Language

The target language $\mathcal{T}$ is based on an addressable memory, a value stack, and operations to load from memory, to store to memory, to push a value onto the stack, and to add the topmost stack elements. Similar to type *variable*, the type *address* is abstract:

```
typedecl   address
types      memory      =  address ⇀ int
types      stack       =  int list
datatype   instruction =  Load address | Store address | Add | Push int
types      Tprogram    =  instruction list
types      store       =  stack × memory
```

For stacks, we assume the functions `top`, `push`, and `pop` with their usual meaning. Execution is defined as follows:

```
execi :: instruction ⇒ store ⇒ store
 execi (Load a)  (stck, mem)          = (push (the (mem a)) stck, mem)
 execi (Store a) (stck, mem)          = (pop stck, mem(a ↦ (top stck)))
 execi IAdd      (i1#i2#rest, mem)    = (push (i1 + i2) rest, mem)
 execi (Push i)  (stck, mem)          = (push i stck, mem)
Exec :: Tprogram ⇒ store ⇒ store
 Exec []     s = s
 Exec i#is s = Exec is (execi i s)
```

2.3 Formalization of Translation Correctness

For modern high-level programming languages, a formalization of translation correctness is a non-trivial task. In particular, it is not sufficient to look at the initial and final states. One has to precisely define what the observable states are and how they are related to the implementation in the target language. In addition, translation correctness has to specify how bounded resources must be handled. These aspects are beyond this paper. We focus here on the relation between the states of $\mathcal{S}$ programs and their counterpart in $\mathcal{T}$ programs.

A memory map maps variables to addresses. A memory `m` and a state `s` are called *conform* w.r.t. a memory map `mu`, if the map is bijective for the elements on which `m` and `s` are defined. Of course, in practical scenarios, memory maps become more complex and the definition of conformance can be less restrictive.

```
types MemMap = variable ⇀ address
conf :: memory ⇒ state ⇒ MemMap ⇒ bool
conf m s mu ≡ ( ∀v ∈ (dom s) . (m ∘ mu) v = s v ) ∧
              ( ∀a ∈ (dom m) . (s ∘ mu⁻¹) a = m a )
```

A $\mathcal{T}$ program `T` is a correct tranlation of an $\mathcal{S}$ program `S`, iff there exists a memory map `mu` such that for all appropriate initial states `st` of `S` the following holds: Starting computation of `S` in `st` yields the same result as executing `T` with an initial memory `m` that is conform to `st` w.r.t. `mu`. With our restrictive version of conformance, the initial memory `m` is uniquely defined by `st` and mu, that is, it can be defined as a function of st and mu and conformance can be proved:

$$\texttt{init\_memory} :: \mathit{state} \Rightarrow \mathit{MemMap} \Rightarrow \mathit{memory}$$
$$\texttt{init\_memory s mu} \equiv \lambda\, \texttt{a} \,.\, \texttt{s} \circ \texttt{mu}^{-1} \circ (\texttt{Some a})$$

Lemma 2.1

$$\forall\, \texttt{mu s} \,.\, (\texttt{dom mu}) = (\texttt{dom s}) \longrightarrow \texttt{conf (init\_memory s mu) s mu}$$

□

Based on these definitions, we can specify the translation predicate:

$$\begin{array}{l} \texttt{ctrans} :: \mathit{Sprogram} \Rightarrow \mathit{Tprogram} \Rightarrow \mathit{bool} \\ \texttt{ctrans S T} \equiv \exists\, \texttt{mu} \,.\, \forall\, \texttt{st} \in (\texttt{appr S}) \,. \\ \qquad\qquad (\; (\texttt{dom mu}) = (\texttt{dom st}) \;) \wedge \\ \qquad\qquad (\; \texttt{Comp st S} = \texttt{top}\,(\texttt{fst}\,(\texttt{Exec T}\,([], \texttt{init\_memory st mu}))) \;) \end{array}$$

Notice that a proof generating compiler knows which memory map μ was used in the translation of a program. It can make use of this knowledge for the generation of the proof. More generally, whenever a proof generating compiler computes certain information I and takes decisions based on I, it can use I to generate a witness to prove an existential property.

3 Generating Correctness Proofs

We built a simple non-optimizing compiler that translates the assignments of source programs to sequences of machine instructions. This translation is illustrated by Fig. 2 for a simple example (the information about states and stores is used later). In this section, we demonstrate how a direct correctness proof based on symbolic evaluation of the programs can be expressed in our framework. This simple proof technique was chosen to explain (a) what proof scripts look like and (b) how compilers can generate proofs based on analysis information they produce. A more realistic proof technique is presented in the next section.

Proof Scripts

For the correctness proofs, we need a representation of source and target programs within the SVF. As already mentioned above, we use abstract syntax here. Instead of using strings for variable identifiers and integers for addresses, our compiler generates the enumeration types *variable_* for the finite number of variables of a source program S and *address_* for the set of needed adresses. This is not necessary, but makes the proofs technically simpler. The enumeration types are embedded by the functions var_ and addr_ into the abstract types *variable* and *address* defined above. We use the abbreviation constructs of Isabelle/HOL to suppress the embedding functions, for example we write

$\mathtt{S}_{\mathtt{ex}}$	environments of $\mathtt{S}_{\mathtt{ex}}$	$\mathtt{T}_{\mathtt{ex}}$	environments of $\mathtt{T}_{\mathtt{ex}}$
	$\mathtt{state}_0$		$\mathtt{store}_0$
[0] $\mathtt{v}_0 = 7 + \mathtt{v}_0$;		[0] Push 7	$\mathtt{store}_1$
		[1] Load $\mathtt{a}_0$	$\mathtt{store}_2$
		[2] Add	$\mathtt{store}_3$
	$\mathtt{state}_1$	[3] Store $\mathtt{a}_0$	$\mathtt{store}_4$
[1] $\mathtt{v}_1 = 8$;		[4] Push 8	$\mathtt{store}_5$
	$\mathtt{state}_2$	[5] Store $\mathtt{a}_1$	$\mathtt{store}_6$
[2] $\mathtt{v}_0 + \mathtt{v}_1$;		[6] Load $\mathtt{a}_0$	$\mathtt{store}_7$
		[7] Load $\mathtt{a}_1$	$\mathtt{store}_8$
		[8] Add	$\mathtt{store}_9$

Fig. 2. An $\mathcal{S}$ program, its translation, and a sketch of the executions.

$\mathtt{v}_0$ for $\mathtt{var\_}\ \mathtt{v}_{0\_}$. Furthermore, the compiler generates an abbreviation for the used memory map. For the example program of Fig. 2, we get the following definitions:

$$\begin{array}{llll}
\textbf{datatype} & \textit{variable\_} & = & \mathtt{v}_{0\_} \mid \mathtt{v}_{1\_} \\
\textbf{datatype} & \textit{address\_} & = & \mathtt{a}_{0\_} \mid \mathtt{a}_{1\_} \\
\textbf{consts} & \mathtt{var\_} & :: & \textit{variable\_} \Rightarrow \textit{variable} \\
\textbf{consts} & \mathtt{addr\_} & :: & \textit{address\_} \Rightarrow \textit{address}
\end{array}$$

$$\begin{array}{lll|lll}
\textbf{syntax} & & & \textbf{translations} & & \\
"\mathtt{v}_0" & :: & \textit{variable} & "\mathtt{v}_0" & == & "\mathtt{var\_}\ \mathtt{v}_{0\_}" \\
"\mathtt{v}_1" & :: & \textit{variable} & "\mathtt{v}_1" & == & "\mathtt{var\_}\ \mathtt{v}_{1\_}" \\
"\mathtt{a}_0" & :: & \textit{address} & "\mathtt{a}_0" & == & "\mathtt{addr\_}\ \mathtt{a}_{0\_}" \\
"\mathtt{a}_1" & :: & \textit{address} & "\mathtt{a}_1" & == & "\mathtt{addr\_}\ \mathtt{a}_{1\_}"
\end{array}$$

$$\begin{array}{lll}
\mu & \equiv & \mathtt{empty}\,(\mathtt{v}_0 \mapsto \mathtt{a}_0)\,(\mathtt{v}_1 \mapsto \mathtt{a}_1) \\
\mathtt{S}_{\mathtt{ex}} & \equiv & (\ [\mathtt{v}_0 ::= (\mathtt{VAR}\ 7) \oplus (\mathtt{VAR}\ \mathtt{v}_0),\ \mathtt{v}_1 ::= (\mathtt{INT}\ 8)]\ ,\ (\mathtt{VAR}\ \mathtt{v}_0)\ \oplus\ (\mathtt{VAR}\ \mathtt{v}_1)\ ) \\
\mathtt{T}_{\mathtt{ex}} & \equiv & [\mathtt{Push}\ 7, \mathtt{Load}\ \mathtt{a}_0, \mathtt{Add}, \mathtt{Store}\ \mathtt{a}_0, \mathtt{Push}\ 8, \mathtt{Store}\ \mathtt{a}_1, \mathtt{Load}\ \mathtt{a}_0, \mathtt{Load}\ \mathtt{a}_1, \mathtt{Add}]
\end{array}$$

The proof script itself consists of a list of lemmas with proofs where a proof may use other lemmas in an acyclic way. For our simple scenario, the compiler generates the six auxiliary lemmas shown in Fig. 3 and the main lemma stating $\mathtt{ctrans}\ \mathtt{S}_{\mathtt{ex}}\ \mathtt{T}_{\mathtt{ex}}$ together with their proofs. Here, we describe the basic idea of the underlying proof and show how such a proof looks like for the example program $\mathtt{S}_{\mathtt{ex}}$.

As the language $\mathcal{S}$ does not support loops and recursion, the compiler can perform a symbolic evaluation of the program yielding an expression $\mathtt{E}$ that only depends on the input variables, that is, on the initial value of those variables that are used in the program before they are defined. For $\mathtt{S}_{\mathtt{ex}}$, expression $\mathtt{E}$ is $(\mathtt{i}_0\ +\ 15)$ where $\mathtt{i}_0$ denotes the initial value of $\mathtt{v}_0$. Altogether,

Lemma 3.1 `mu_v0_EQ_a0`

$(\mathtt{the} \circ \mu)\ \mathtt{v}_0 \ = \ \mathtt{a}_0$ □

Lemma 3.2 `mu_v1_EQ_a1`

$(\mathtt{the} \circ \mu)\ \mathtt{v}_1 \ = \ \mathtt{a}_1$ □

Lemma 3.3 `inv_mu_a0_EQ_v0`

$(\mu^{-1} \circ \mathtt{Some})\ \mathtt{a}_0 \ = \ \mathtt{v}_0$ □

Lemma 3.4 `inv_mu_a1_EQ_v1`

$(\mu^{-1} \circ \mathtt{Some})\ \mathtt{a}_1 \ = \ \mathtt{v}_1$ □

Lemma 3.5

$\mathtt{s} \in (\mathtt{appr}\ \mathtt{S}_{\mathtt{ex}}) \ \longrightarrow \ (\exists\, \mathtt{i}_0 \,.\, (\mathtt{s}\ \mathtt{v}_0) \ = \ (\mathtt{Some}\ \mathtt{i}_0)) \wedge (\exists\, \mathtt{i}_1 \,.\, (\mathtt{s}\ \mathtt{v}_1) \ = \ (\mathtt{Some}\ \mathtt{i}_1))$ □

Lemma 3.6

$(\mathtt{s} \in (\mathtt{appr}\ \mathtt{S}_{\mathtt{ex}}) \ \longrightarrow \ \mathtt{conf}\ (\mathtt{init\_memory}\ \mathtt{s}\ \mu)\ \mathtt{s}\ \mu$ □

Fig. 3. Auxiliary lemmas

the correctness proof of $\mathtt{ctrans}\ \mathtt{S}_{\mathtt{ex}}\ \mathtt{T}_{\mathtt{ex}}$ becomes the following form:

Proof of the main lemma for $\mathtt{S}_{\mathtt{ex}}$ and $\mathtt{T}_{\mathtt{ex}}$:

$\mathtt{ctrans}\ \mathtt{S}_{\mathtt{ex}}\ \mathtt{T}_{\mathtt{ex}}$

$=$ [by the definition of `ctrans`]

$\exists\, \mathtt{mu} \,.\, \forall\, \mathtt{st} \in (\mathtt{appr}\ \mathtt{S}_{\mathtt{ex}}) \,.$
$(\ \mathtt{conf}\ (\mathtt{init\_memory}\ \mathtt{st}\ \mathtt{mu})\ \mathtt{st}\ \mathtt{mu} \ ) \wedge$
$(\ \mathtt{Comp}\ \mathtt{S}_{\mathtt{ex}}\ \mathtt{T}_{\mathtt{ex}} \ = \ \mathtt{top}\,(\mathtt{fst}\,(\mathtt{Exec}\ \mathtt{T}_{\mathtt{ex}}\ ([], \mathtt{init\_memory}\ \mathtt{st}\ \mathtt{mu})))\ )$

$\Longleftarrow$ [application of the rule $(\exists\mathtt{I})$ instantiating rules $\exists -$ variable $\mathtt{x}$ by μ]

$\forall\, \mathtt{st} \in (\mathtt{appr}\ \mathtt{S}_{\mathtt{ex}}) \,.$
$(\ \mathtt{conf}\ (\mathtt{init\_memory}\ \mathtt{st}\ \mu)\ \mathtt{st}\ \mu \ ) \wedge$
$(\ \mathtt{Comp}\ \mathtt{S}_{\mathtt{ex}}\ \mathtt{T}_{\mathtt{ex}} \ = \ \mathtt{top}\,(\mathtt{fst}\,(\mathtt{Exec}\ \mathtt{T}_{\mathtt{ex}}\ ([], \mathtt{init\_memory}\ \mathtt{st}\ \mu)))\ )$

$\Longleftarrow$ [application of the rule $\mathtt{bounded} - (\forall\mathtt{I})$. This introduces fresh free variable `s` of type *state* and a new assumption into the proof state]

Assumption : $\mathtt{s} \in (\mathtt{appr}\ \mathtt{S}_{\mathtt{ex}})$
Subgoal : $\mathtt{conf}\ (\mathtt{init\_memory}\ \mathtt{s}\ \mu)\ \mathtt{s}\ \mu \wedge$
$(\mathtt{Comp}\ \mathtt{s}\ \mathtt{S}_{\mathtt{ex}}) \ = \ \mathtt{top}\,(\mathtt{fst}\,(\mathtt{Exec}\ \mathtt{T}_{\mathtt{ex}}\ ([], \mathtt{init\_memory}\ \mathtt{s}\ \mu)))$

$\Longleftarrow$ [application of the rule $\mathtt{bounded} - (\wedge\mathtt{I})$. This splits the current proof goal in two separate subgoals]

Assumption : $\mathtt{s} \in (\mathtt{appr}\ \mathtt{S}_{\mathtt{ex}})$
Subgoal 1 : $\mathtt{conf}\ (\mathtt{init\_memory}\ \mathtt{s}\ \mu)\ \mathtt{s}\ \mu$
Subgoal 2 : $(\mathtt{Comp}\ \mathtt{s}\ \mathtt{S}_{\mathtt{ex}}) \ = \ \mathtt{top}\,(\mathtt{fst}\,(\mathtt{Exec}\ \mathtt{T}_{\mathtt{ex}}\ ([], \mathtt{init\_memory}\ \mathtt{s}\ \mu)))$

Subgoal 1 is a direct consequence of Lemma 3.6. To prove Subgoal 2, we introduce new assumptions reasoning in a forward manner:

$\textit{Assumption} \;:\; \texttt{s} \in (\texttt{appr}\ \texttt{S}_{\texttt{ex}})$
$\textit{Subgoal 2} \;:\; (\texttt{Comp s S}_{\texttt{ex}}) \;=\; \texttt{top}\,(\texttt{fst}\,(\texttt{Exec T}_{\texttt{ex}}\ ([],\ \texttt{init\_memory s}\ \mu)))$

$\Rightarrow$ [application of the Lemma 3.5 and unfolding of the definition of `appr`]

$\textit{Assumption} \;:\; \texttt{s} \in \{\texttt{s}' \mid (\texttt{Vars S}_{\texttt{ex}}) \;=\; (\texttt{dom s}')\},$
$(\exists\, \texttt{i}_0\ .\ \texttt{s v}_0 = \texttt{Some i}_0) \;\wedge\; (\exists\, \texttt{i}_1\ .\ \texttt{s v}_1 = \texttt{Some i}_1)$
$\textit{Subgoal 2} \;:\; (\texttt{Comp s S}_{\texttt{ex}}) \;=\; \texttt{top}\,(\texttt{fst}\,(\texttt{Exec T}_{\texttt{ex}}\ ([],\ \texttt{init\_memory s}\ \mu)))$

$\Rightarrow$ [$(\texttt{Vars S}_{\texttt{ex}}) \;=\; \{\texttt{v}_0,\ \texttt{v}_1\}$, application of the rules : $(\wedge E_1)$, $(\wedge E_2)$ and $\exists E$ twice introduces two fresh free variables $\texttt{i}_0$ and $\texttt{i}_1$ of type *int*]

$\textit{Assumption} : (\texttt{s} = \texttt{empty}(\texttt{v}_0 \mapsto \texttt{i}_0)(\texttt{v}_1 \mapsto \texttt{i}_1)), (\texttt{s v}_0 = \texttt{Some i}_0), (\texttt{s v}_1 = \texttt{Some i}_1)$
$\textit{Subgoal 2} \;:\; (\texttt{Comp s S}_{\texttt{ex}}) \;=\; \texttt{top}\,(\texttt{fst}\,(\texttt{Exec T}_{\texttt{ex}}\ ([],\ \texttt{init\_memory s}\ \mu)))$

Now, we make use of the result of the symbolic evaluation. By application of the transitivity rule, Subgoal 2 is split into two separate subgoals in which the result of the symbolic evaluation $(\texttt{i}_0\ +\ 15)$ is used as result of the computation of the source program and as result of the execution of the target program.

$\textit{Assumption} : (\texttt{s} = \texttt{empty}(\texttt{v}_0 \mapsto \texttt{i}_0)(\texttt{v}_1 \mapsto \texttt{i}_1)), (\texttt{s v}_0 = \texttt{Some i}_0), (\texttt{s v}_1 = \texttt{Some i}_1)$
$\textit{Subgoal 2.1} : (\texttt{Comp s S}_{\texttt{ex}}) \;=\; (\texttt{i}_0 + 15)$
$\textit{Subgoal 2.2} : (\texttt{i}_0 + 15) \;=\; \texttt{top}\,(\texttt{fst}\,(\texttt{Exec T}_{\texttt{ex}}\ ([],\ \texttt{init\_memory s}\ \mu)))$

The proof of the subgoal 2.1 is straightforward and it succeeds after unfolding the definition of `Comp` and $\texttt{S}_{\texttt{ex}}$, substitution of `s` by $\texttt{empty}(\texttt{v}_0 \mapsto \texttt{i}_0)(\texttt{v}_1 \mapsto \texttt{i}_1)$ in the left-hand side of the equation and rewriting. To prove the Subgoal 2.2, we unfold the definition of `init_memory`:

$\textit{Subgoal 2.2} \;:\; (\texttt{i}_0 + 15) \;=\; \texttt{top}\,(\texttt{fst}\,(\texttt{Exec T}_{\texttt{ex}}\ ([],\ \lambda\,\texttt{a}\ .\ \texttt{s}\,(\mu^{-1}\,(\texttt{Some a})))))$

By the definition, μ is a bijection with domain $\{\texttt{v}_0,\ \texttt{v}_1\}$ and range $\{\texttt{a}_0,\ \texttt{a}_1\}$. Thus, μ^{-1} is a bijection with domain $\{\texttt{a}_0,\ \texttt{a}_1\}$ and range $\{\texttt{v}_0,\ \texttt{v}_1\}$. Using the assumption $(\texttt{s} \;=\; \texttt{empty}(\texttt{v}_0 \mapsto \texttt{i}_0)(\texttt{v}_1 \mapsto \texttt{i}_1))$ and applying Lemmas `inv_mu_a0_EQ_v0` and `inv_mu_a1_EQ_v1`, the term $\lambda\,\texttt{a}\ .\ \texttt{s}\,(\mu^{-1}\,(\texttt{Some a}))$ in the rigth-hand side of Subgoal 2.2 can be rewriten to $\texttt{empty}(\texttt{a}_0 \mapsto \texttt{i}_0)(\texttt{a}_1 \mapsto \texttt{i}_1)$. The proof of the equation

$\textit{Subgoal 2.2} \;:\; (\texttt{i}_0 + 15) \;=\; \texttt{top}\,(\texttt{fst}\,(\texttt{Exec T}_{\texttt{ex}}\ ([],\ \texttt{empty}(\texttt{a}_0 \mapsto \texttt{i}_0)(\texttt{a}_1 \mapsto \texttt{i}_1))))$

is then straightforward and it succeeds after unfolding the definition of `Exec` and $\texttt{T}_{\texttt{ex}}$ and rewriting.

Let us stress again that we explained the above proof technique only to show in some detail how generated proofs look like and how information computed by the compiler – here the memory map and the result of the symbolic evaluation – can be used for proof generation. For practical proof generating compilers, the proof technique has two disadvantages: (a) It is not applicable to realistic programs with loops. (b) Checking the resulting proofs is slow as it incorporates evaluation by rewriting of source and target program. The interesting aspect here is that efficiency of proof checking is an issue for practical scenarios.

Proof Generation
Several parts of the above proof are program dependent, in particular: the term ($\mathtt{i}_0 + \mathtt{15}$) and most of the auxiliary lemmas. Their generation is performed by the instrumentation of the compiler. The instrumentation is based on well understood compilation techniques und is used for optimization, symbolic evaluation and unparsing.

4 Pattern-based Correctness Proofs

In the last section, we concentrated on the principle aspects of our approach. In particular, we explained how programs and proofs are represented. For illustration reasons, we used a simple proof technique based on symbolic evaluation. For more complex languages, such a technique is not applicable. In this section, we describe a more realistic proof technique and use it to demonstrate the flexibility of the overall approach. Whereas the first technique was directly based on the semantics of source and target program, the second technique is closer to verification of compilation algorithms. The flexibility to enable different proof techniques is important. This way, different steps in a compiler architecture can be handled by appropriate proof techniques. In particular, we can exploit the proof and reasoning techniques developed in [7] and implement them in our SVF.

The central requirement of our approach is to prove `ctrans S T`. The approach does not rely on a particular proof technique. The proofs might depend in their structure and arguments on `S` and `T`. They might as well consist of a large fixed part that is independent of `S` and `T`, and a small part depending on `S` and `T`. In this section, we illustrate a proof technique according to the latter scheme. It is based on a program independent lemma stating that whenever source and target programs `S` and `T` satisfy a certain, easily checkable *syntactical* translation relation, `T` is a correct translation of `S`:

Lemma 4.1

$$(\exists \mathtt{mu}\ .\ \mathtt{tr\ S\ mu\ T}\ ) \quad \longrightarrow \quad \mathtt{ctrans\ S\ T}$$

□

The specification of `tr`, the lemma, and its proof are specification parts specific to the compiler (cf. Fig. 1). They are program independent and are developed together with the proof generating compiler. Of course, the proof of Lemma 4.1 has to be available in the SVF. Based on these specification parts and the proof of the lemma, the generated translation correctness proof for a source program `S` consists of a proof of ($\exists$ `mu . tr S mu T`) and an application of the lemma. Notice that μ and `T` are known by the compiler. The basic idea of this technique is that proving a syntactical relation between `S` and `T` is simpler than showing semantical equivalence. To make this more clear, we illustrate the technique for the source and target language of Sect. 2.

The first step is to provide the program independent definition of tr. In our simple scenario, we use translation functions for the different language constructs following the scheme of Fig. 2.

$\texttt{tre} :: \textit{expression} \Rightarrow \textit{MemMap} \Rightarrow \textit{instruction list}$
$\texttt{tre (INT i)} \quad \texttt{mu} = [\texttt{Push i}]$
$\texttt{tre (VAR v)} \quad \texttt{mu} = [\texttt{Load (mu v)}]$
$\texttt{tre (e1} \oplus \texttt{e2) mu} = (\texttt{tre e1 mu}) @ (\texttt{tre e2 mu}) @ [\texttt{Add}]$

$\texttt{tra} :: \textit{assignment} \Rightarrow \textit{MemMap} \Rightarrow \textit{instruction list}$
$\texttt{tra (v ::= e) mu} = (\texttt{tre e mu}) @ [\texttt{ILoad (mu v)}]$

$\texttt{trass} :: \textit{assignment list} \Rightarrow \textit{MemMap} \Rightarrow \textit{instruction list}$
$\texttt{trass []} \quad \texttt{mu} = []$
$\texttt{trass (a\#as) mu} = (\texttt{tra a mu}) @ (\texttt{trass as mu})$

$\texttt{trp} :: \textit{Sprogram} \Rightarrow \textit{MemMap} \Rightarrow \textit{instruction list}$
$\texttt{trp (ass, e) mu} = (\texttt{trass ass mu}) @ (\texttt{tre e mu})$

$\texttt{tr} :: \textit{Sprogram} \Rightarrow \textit{MemMap} \Rightarrow \textit{Tprogram} \Rightarrow \textit{bool}$
$\texttt{tr S mu T} = (\texttt{trp S mu}) = \texttt{T}$

For our simple languages, the definition of the transition relation almost specifies a translation algorithm. In more realistic scenarios, a translation relation need not be a function and can as well capture optimization steps.

Given programs S, T, and mu, the compiler has to generate a proof for tr S mu T. This proof consists of a straightforward unfolding of the specifications, that is, the proof generating part of the compiler becomes very simple. We illustrate this by the proof that is generated for our example program $\texttt{S}_{\texttt{ex}}$, that is, we show $\texttt{trp}\ \texttt{S}_{\texttt{ex}}\ \mu = \texttt{T}_{\texttt{ex}}$:

$\texttt{trp}\ \texttt{S}_{\texttt{ex}}\ \mu$
$= \quad [[\text{the definition of } \texttt{trp}]]$
$(\texttt{trass}\ [\texttt{v}_0 = \texttt{v}_0 + 7,\ \texttt{v}_1 = 8]\ \mu) @ (\texttt{tre}\ (\texttt{v}_0 + \texttt{v}_1)\ \mu)$
$= \quad [[\text{the definition of } \texttt{trass}]]$
$(\texttt{tra}\ (\texttt{v}_0 = \texttt{v}_0 + 7)\ \mu) @ (\texttt{trass}\ [\texttt{v}_1 = 8]\ \mu) @ (\texttt{tre}\ (\texttt{v}_0 + \texttt{v}_1)\ \mu)$
$= \quad [[\text{the definition of } \texttt{tra}]]$
$[\texttt{Load}\ \mu\,(\texttt{v}_0),\ \texttt{Push 8},\ \texttt{Add},\ \texttt{Store}\ \mu\,(\texttt{v}_0)] @ (\texttt{trass}\ [\texttt{v}_1 = 8]\ \mu) @ (\texttt{tre}\ (\texttt{v}_0 + \texttt{v}_1)\ \mu)$
$= \quad [[\text{the definition of } \mu:\ \ \mu \equiv \texttt{empty}(\texttt{v}_0 \mapsto \texttt{a}_0)(\texttt{v}_1 \mapsto \texttt{a}_1)]]$
$[\texttt{Load a}_0,\ \texttt{Push 8},\ \texttt{Add},\ \texttt{Store a}_0] @ (\texttt{trass}\ [\texttt{v}_1 = 8]\ \mu) @ (\texttt{tre}\ (\texttt{v}_0 + \texttt{v}_1)\ \mu)$
$= \quad [[\text{the definition of } \texttt{trass}]]$
$[\texttt{Load a}_0,\ \texttt{Push 8},\ \texttt{Add},\ \texttt{Store a}_0] @ (\texttt{tra}\ (\texttt{v}_1 = 8)\ \mu) @ (\texttt{trass}\ []\ \mu)$
$@ (\texttt{tre}\ (\texttt{v}_0 + \texttt{v}_1)\ \mu)$
$= \quad [[\text{the definition of } \texttt{tra}]]$
$[\texttt{Load a}_0,\ \texttt{Push 8},\ \texttt{Add},\ \texttt{Store a}_0] @ [\texttt{Push 8},\ \texttt{Store}\ \mu\,(\texttt{v}_1)] @ [] @ (\texttt{tre}\ (\texttt{v}_0 + \texttt{v}_1)\ \mu)$
$= \quad [[\text{the definition of } \mu:\ \ \mu \equiv \texttt{empty}(\texttt{v}_0 \mapsto \texttt{a}_0)(\texttt{v}_1 \mapsto \texttt{a}_1)]]$
$[\texttt{Load a}_0,\ \texttt{Push 8},\ \texttt{Add},\ \texttt{Store a}_0] @ [\texttt{Push 8},\ \texttt{Store a}_1] @ [] @ (\texttt{tre}\ (\texttt{v}_0 + \texttt{v}_1)\ \mu)$
$=$
$[\texttt{Load a}_0,\ \texttt{Push 8},\ \texttt{Add},\ \texttt{Store a}_0] @ [\texttt{Push 8},\ \texttt{Store a}_1] @ (\texttt{tre}\ (\texttt{v}_0 + \texttt{v}_1)\ \mu)$

$=$ [[the definition of $\texttt{tre}$]]
[$\texttt{Load a}_0$, $\texttt{Push 8}$, $\texttt{Add}$, $\texttt{Store a}_0$]@[$\texttt{Push 8}$, $\texttt{Store a}_1$]
@[$\texttt{Load}\ \mu(\texttt{v}_0)$, $\texttt{Load}\ \mu(\texttt{v}_1)$, $\texttt{Add}$]
$=$ [[the definition of μ : $\mu \equiv \texttt{empty}(\texttt{v}_0 \mapsto \texttt{a}_0)(\texttt{v}_1 \mapsto \texttt{a}_1)$]]
[$\texttt{Load a}_0$, $\texttt{Push 8}$, $\texttt{Add}$, $\texttt{Store a}_0$]@[$\texttt{Push 8}$, $\texttt{Store a}_1$]@[$\texttt{Load a}_0$, $\texttt{Load a}_1$, $\texttt{Add}$]
$=$
[$\texttt{Load a}_0$, $\texttt{Push 8}$, $\texttt{Add}$, $\texttt{Store a}_0$, $\texttt{Push 8}$, $\texttt{Store a}_1$, $\texttt{Load a}_0$, $\texttt{Load a}_1$, $\texttt{Add}$]
$=$ [[the definition of $\texttt{T}_{\texttt{ex}}$]]
$\texttt{T}_{\texttt{ex}}$

Notice that the size of the resulting proof is linear w.r.t. to the length of the input program.

5 Conclusions

We presented an approach to compilers that, given source program S, generate a target program T together with a formal proof that T is a correct translation of S. Whereas most work in this area concentrated on logics and proof techniques for such proofs, this presentation discussed the issues on how existing formal specification and verification frameworks can be used as a basis for such proof generating compilers. In particular, we introduced the notion of *compiler-independent* translation contracts and showed how additional *compiler-dependent* specification parts can be helpful. A clear separation of these aspects is necessary to provide

- appropriate language specifications that are not tailored towards compiler issues and
- the flexibility to adapt the proof tasks to the compiler architecture.

An important aspect of the approach is that the underlying general SVF allows to use and combine different proof techniques.

To demonstrate our approach, we implemented a proof generating compiler in ML that translates a simple assignment language into a stack machine language. The mentioned flexibility was illustrated by developing two very different, simple proof techniques: One based on symbolic evaluation, the other one based on syntactic program patterns. As SVF, we used Isabelle/HOL. Currently, we work on simple optimizations. As our next step, we plan to implement translation and proof techniques as described in [7] within our framework. Furthermore, we aim to improve the proof checking support.

References

[1] Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science*, 2001.

[2] Gerhard Goos and Wolf Zimmermann. Verification of compilers. In *Correct System Design*, pages 201–230, 1999.

[3] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[4] George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[5] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[6] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, New York, NY, USA, 1994.

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

[8] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT Laboratory for Computer Science, March 1999.

[9] Martin Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.

[10] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for translation validation of optimizing compilers.

Appendix

5.1 Isabelle/HOL

In this section, we present summarizations of two predefined Isabelle/HOL theories: `Option` and `Map` that we used to formalize the languages $\mathcal{S}$ and $\mathcal{T}$. The following two sections are adapted quotations of Sections 3.4.3 and 3.4.4 from Nipkow et al. [5].

5.1.1 Options

The datatype of optional values

$$\begin{array}{lll} \textbf{datatype} & \alpha\ \textit{option} & = \texttt{None} \mid \texttt{Some}\ \alpha \\ \textbf{consts} & \texttt{the}\ ::\ \alpha\ \textit{option} & \rightarrow\ \alpha \\ \textbf{primrec} & \texttt{the}\,(\texttt{Some x}) & =\ \texttt{x} \\ & \texttt{the None} & =\ \texttt{arbitrary} \end{array}$$

is used to add a new element `None` to a type and wrap the remaining elements up in `Some`; function `the` unwraps them again; on `None` it can be defined arbitrarily.

5.1.2 Mappings

One frequently needs partial functions where one can determine if an entry is defined or not. We call them mappings and define them as functions with optional range type:

types $\alpha \rightsquigarrow \beta \;=\; \alpha \;\rightarrow\; \beta \; \mathit{option}$

Typical applications are symbol tables (where declared names are mapped to some information) or heap storage (where allocated addresses are mapped to their content).

For the convenient manipulation of mappings the following functions are provided:

$$\begin{array}{lll} \texttt{empty} & :: & \alpha \rightarrow \beta \\ \_(\_ \mapsto \_) & :: & (\alpha \rightsquigarrow \beta) \rightarrow \alpha \rightarrow \beta \rightarrow (\alpha \rightsquigarrow \beta) \end{array}$$

They represent the empty mapping and updating of a mapping in one place.