Electronic Notes in Theoretical Computer Science 132 (2005) 37–51 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Towards Proof Generating Compilers

# Arnd Poetzsch-Heffter [a](#_bookmark0),[1](#_bookmark1) Marek Gawkowski [a](#_bookmark0),[2](#_bookmark1)

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. Specifi- cation 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.

*Keywords:* compilation correctness, translation validation, formal translation contract, automatic proof generation, proof checking

# 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 devel- opment, 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

1 Email: [poetzsch@informatik.uni-kl.de](mailto:poetzsch@informatik.uni-kl.de)

2 Email: [gawkowsk@informatik.uni-kl.de](mailto:gawkowsk@informatik.uni-kl.de)

1571-0661 © 2005 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2005.03.023

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. 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](#_bookmark11)]):
   * 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.
2. 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.
3. Compiler implementation correctness: For compilation correctness, it is not sufficient to prove that the high-level program that describes the com- piler is correct. One has to show as well that this program is translated correctly and runs on a correct system.

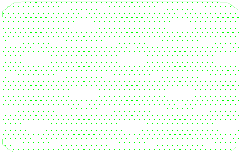
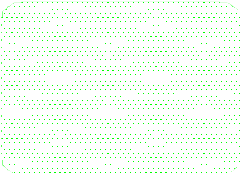
To avoid the last problem, many researchers in the area have concentrated on *translation correctness* in the last years. The basic difference between com- piler correctness and translation correctness is as follows: instead of proving that the compiler implementation correctly translates all admissible 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](#_bookmark17)], 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](#_bookmark2) 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,

source program S target program T



proof generating compiler

translation contract: specific to

specification & verification framework

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

compiler implementation:

further specifica- tion parts, e.g.:

* intermediate languages
* analysis forma- lizations

derived properties

translation correct: yes/no

Fig. 1. Proof generating compilation and checking

* + 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).

Our approach to translation correctness is characterized by (a) a clear sepa- ration between the two language specifications, (b) an explicit specification of translation correctness, (c) the flexibility gained by supporting further speci- fications and deriving program independent properties from the given specifi- cations, 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 spec- ifications 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 suf- ficiently 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](#_bookmark16)]) 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 specifi- cations 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](#_bookmark2)). However, the certifying proofs have to show that the translation contract is fulfilled. Alto- gether, a generated proof script provides checkable evidence that a translation is correct.

*Related Work*

Although this work has profited from several publications on compiler correct- ness in general (e.g. [[2](#_bookmark11),[9](#_bookmark18)]), 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](#_bookmark19),[8](#_bookmark17)]. 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 pow- erful 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 *trans- lation validation* try to avoid this and focus on automating the validation pro- cess 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](#_bookmark20)], the background of a validation tool VOC-64 for optimizations of the SGI Pro-64 compiler is described. Sim- ilar 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 ver-

ification conditions. Necula [[4](#_bookmark14)] 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](#_bookmark11)]: Instead of proving the usually complex translation algorithm correct, it adds a result checking procedure to the compiler that is run after translation. A com- pilation 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](#_bookmark13)], Necula and Lee described certifying compilers. A certifying com- piler 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 depend- ing 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 encoding of logical frameworks, see [[1](#_bookmark12)]).

*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](#_bookmark3), we formally specify the translation contract. Sections [3](#_bookmark4) and [4](#_bookmark9) 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 cor- rect compilation can be derived if certain syntactical relations between source and target program hold.

# 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, S and T , and of a corresponding translation predicate ctrans. Thus, ctrans S T expresses the fact that T, a program in target language T , is a correct translation of S, a program in source language S, as defined by ctrans.

* 1. *Formalization of the* S *Language*

The programs S of the source language 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 S is specified as follows:

typedecl *variable*

datatype *expression* = INT *int* | VAR *variable* | *expression* ⊕ *expression*

datatype *assignment* = *variable* := *expression*

types *Sprogram* = (*assignment list* ) × *expression*

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 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* ⇒ *α* which has the following definition: *λ* v *.* case v of Some x ⇒ x | None ⇒ arbitrary. See Appendix for the description of Isabelle/HOL formalisms we used here.

types *state* = *variable* ~ *int*

eval :: *expression* ⇒ *state* ⇒ *int*

eval (INT i) s = i

eval (VAR v) s = the (s v)

eval (e1 ⊕ e2 ) s = (eval e1 s) + (eval e2 s)

comp :: *state* × (*assignment list* ) ⇒ *state* × (*assignment list* )

comp (s*,* []) = (s*,* [])

comp (s*,* (v ::= e)#as) = comp (s(v '→ (eval e s))*,* as) Comp :: *state* ⇒ *Sprogram* ⇒ *int*

Comp s (as*,* e) ≡ eval e (fst(comp (s*,* as)))

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:

appr S ≡ {st *.* (Vars S) = (dom st)} *,*

where the auxiliary function Vars :: *Sprogram* ⇒ *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.

* 1. *Formalization of the* T *Language*

The target language 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)

* 1. *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 S programs and their counterpart in 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 = sv ) ∧

−1

*.*

( ∀ a ∈ (dom m) (s ◦ mu ) a = ma )

A T program T is a correct tranlation of an 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:

init memory :: *state* ⇒ *MemMap* ⇒ *memory*

−1

init memory s mu ≡ *λ* a *.* s ◦ mu ◦ (Some a)

Lemma 2.1

∀ mu s *.* (dom mu)= (dom s) −→ conf (init memory s mu) s mu

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

ctrans :: *Sprogram* ⇒ *Tprogram* ⇒ *bool*

ctrans S T ≡ ∃ mu *.* ∀ st ∈ (appr S) *.*

( (dom mu) = (dom st) ) ∧

( Comp st S = top (fst (Exec T ([]*,* init memory st mu))) )

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.

# 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](#_bookmark5) 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 pro- grams 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.

|  |  |  |  |
| --- | --- | --- | --- |
| Sex | states of Sex | Tex | stores of Tex |
|  | state0 |  | store0 |
| [0] v0 = 7 + v0 ; | state1 | 1. Push 7 2. Load a0 3. Add 4. Store a0 | store1 store2 store3  store4 |
| 1. v1 = 8; | state2 | 1. Push 8 2. Store a1 | store5  store6 |
| [2] v0 + v1; |  | 1. Load a0 2. Load a1 3. Add | store7 store8  store9 |

Fig. 2. An S program, its translation, and a sketch of the executions.

This is not necessary, but makes the proofs technically simpler. The enumer- ation 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

v0 for var v0 . Furthermore, the compiler generates an abbreviation for the used memory map. For the example program of Fig. [2](#_bookmark5), we get the following definitions:

datatype *variable* = v0 | v1

datatype *address* = a0 | a1

consts var :: *variable* ⇒ *variable*

consts addr :: *address* ⇒ *address*

syntax translations

”v0 ” :: *variable* ”v0 ” == ”var v0 ”

*.* ”v1 ” :: *variable* ”v1 ” == ”var v1 ”

”a0 ” :: *address* ”a0 ” == ”addr a0 ” ”a1 ” :: *address* ”a1 ” == ”addr a1 ”

*µ* ≡ empty (v0 '→ a0 ) (v1 '→ a1 )

Sex ≡ ( [v0 ::= (VAR 7) ⊕ (VAR v0 )*,* v1 ::= (INT 8)] *,* (VAR v0 ) ⊕ (VAR v1 ) )

Tex ≡ [Push 7*,* Load a0 *,* Add*,* Store a0 *,* Push 8 *,* Store a1 *,* Load a0*,* Load a1 *,* Add]

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](#_bookmark6) and the main lemma stating

ctrans Sex Tex 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 Sex.

As the language S does not support loops and recursion, the compiler

can perform a symbolic evaluation of the program yielding an expression 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 Sex, ex-

pression E is (i0 + 15) where i0 denotes the initial value of v0. Altogether,

Fig. 3. Auxiliary lemmas

(the ◦ *µ*) v0 = a0

(the ◦ *µ*) v1 = a1

Lemma 3.3 inv mu a0 EQ v0

Lemma 3.4 inv mu a1 EQ v1

(*µ*

−1

* Some) a0 = v0

(*µ* ◦ Some) a1 = v1

Lemma 3.5

s ∈ (appr Sex ) −→ (∃ i0 *.* (s v0) = (Some i0 )) ∧ (∃ i1 *.* (s v1) = (Some i1 ))

−1

Lemma 3.6

(s ∈ (appr Sex ) −→ conf (init memory s *µ*) s *µ*

Lemma 3.1 mu v0 EQ a0

Lemma 3.2 mu v1 EQ a1

the correctness proof of ctrans Sex Tex becomes the following form:

Proof of the main lemma for Sex and Tex :

ctrans Sex Tex

= [by the definition of ctrans]

∃ mu *.* ∀ st ∈ (appr Sex ) *.*

( conf (init memory st mu) st mu ) ∧

( Comp Sex Tex = top (fst (Exec Tex ([]*,* init memory st mu))) )

⇐= [application of the rule (∃I) instantiating rules ∃− variable x by *µ*]

∀ st ∈ (appr Sex ) *.*

( conf (init memory st *µ*) st *µ* ) ∧

( Comp Sex Tex = top (fst (Exec Tex ([]*,* init memory st *µ*))) )

⇐= [application of the rule bounded − (∀I)*.* This introduces fresh free variable s of type *state* and a new assumption into the proof state]

*Assumption* : s ∈ (appr Sex )

*Subgoal* : conf (init memory s *µ*) s *µ* ∧

(Comp s Sex ) = top (fst (Exec Tex ([]*,* init memory s *µ*)))

⇐= [application of the rule bounded − (∧I)*.* This splits the current proof goal in two separate subgoals]

*Assumption* : s ∈ (appr Sex )

*Subgoal 1* : conf (init memory s *µ*) s *µ*

*Subgoal 2* : (Comp s Sex ) = top (fst (Exec Tex ([]*,* init memory s *µ*)))

Subgoal 1 is a direct consequence of Lemma [3.6](#_bookmark7). To prove Subgoal 2, we introduce new assumptions reasoning in a forward manner:

*Assumption* : s ∈ (appr Sex )

*Subgoal 2* : (Comp s Sex ) = top (fst (Exec Tex ([]*,* init memory s *µ*)))

⇒ [application of the Lemma [3.5](#_bookmark8) and unfolding of the definition of appr]

'

'

*Assumption* : s ∈ {s

| (Vars Sex ) = (dom s )}*,*

(∃ i0 *.* s v0 = Some i0 ) ∧ (∃ i1 *.* s v1 = Some i1)

*Subgoal 2* : (Comp s Sex ) = top (fst (Exec Tex ([]*,* init memory s *µ*)))

⇒ [(Vars Sex ) = {v0 *,* v1 }*,* application of the rules : (∧E1 )*,* (∧E2 ) and ∃E twice introduces two fresh free variables i0 and i1 of type *int* ]

*Assumption* : (s = empty(v0 '→ i0 )(v1 '→ i1 ))*,* (s v0 = Some i0 )*,* (s v1 = Some i1 )

*Subgoal 2* : (Comp s Sex ) = top (fst (Exec Tex ([]*,* init memory s *µ*)))

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 (i0 + 15) is used as result of the com-

putation of the source program and as result of the execution of the target

program.

*Assumption* : (s = empty(v0 '→ i0 )(v1 '→ i1 ))*,* (s v0 = Some i0 )*,* (s v1 = Some i1 )

*Subgoal 2 .1* : (Comp s Sex ) = (i0 + 15)

*Subgoal 2 .2* : (i0 + 15) = top (fst (Exec Tex ([]*,* init memory s *µ*)))

The proof of the subgoal 2.1 is straightforward and it succeeds after unfolding

the definition of Comp and Sex, substitution of s by empty(v0 '→ i0)(v1 '→ i1) in the left-hand side of the equation and rewriting. To prove the Subgoal 2.2, we unfold the definition of init memory:

−1

*Subgoal 2 .2* : (i0 + 15) = top (fst (Exec Tex ([]*, λ* a *.* s (*µ* (Some a)))))

By the definition, *µ* is a bijection with domain {v0*,* v1} and range {a0*,* a1}. Thus, *µ*−1 is a bijection with domain {a0*,* a1} and range {v0*,* v1}. Using the assumption (s = empty(v0 '→ i0)(v1 '→ i1)) and applying Lemmas inv mu − a0 EQ v0 and inv mu a1 EQ v1, the term *λ* a *.* s (*µ*−1 (Some a)) in the rigth- hand side of Subgoal 2.2 can be rewriten to empty(a0 '→ i0)(a1 '→ i1). The proof of the equation

*Subgoal 2 .2* : (i0 + 15) = top (fst (Exec Tex ([]*,* empty(a0 '→ i0 )(a1 '→ i1 ))))

is then straightforward and it succeeds after unfolding the definition of Exec

and Tex 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 com- puted 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 inter- esting 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 (i0 + 15) and most of the auxiliary lemmas. Their generation is per-

formed by the instrumentation of the compiler. The instrumentation is based

on well understood compilation techniques und is used for optimization, sym- bolic evaluation and unparsing.

# 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 eval- uation. For more complex languages, such a technique is not applicable. In this section, we describe a more realistic proof technique and use it to demon- strate 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](#_bookmark19)] 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

(∃ mu *.* tr S mu T ) −→ ctrans S T

The specification of tr, the lemma, and its proof are specification parts specific to the compiler (cf. Fig. [1](#_bookmark2)). They are program independent and are developed together with the proof generating compiler. Of course, the proof of Lemma [4.1](#_bookmark10) 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 (∃ mu *.* trS muT ) 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](#_bookmark3).

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](#_bookmark5).

tre :: *expression* ⇒ *MemMap* ⇒ *instruction list*

tre (INT i) mu = [Push i]

tre (VAR v) mu = [Load (mu v)]

tre (e1 ⊕ e2) mu = (tre e1 mu)@ (tre e2 mu)@[Add]

tra :: *assignment* ⇒ *MemMap* ⇒ *instruction list*

tra (v ::= e) mu = (tre e mu)@ [ILoad (mu v)]

trass :: *assignment list* ⇒ *MemMap* ⇒ *instruction list*

trass [] mu = []

trass (a#as) mu = (tra a mu)@ (trass as mu)

trp :: *Sprogram* ⇒ *MemMap* ⇒ *instruction list*

trp (ass*,* e) mu = (trass ass mu)@ (tre e mu)

tr :: *Sprogram* ⇒ *MemMap* ⇒ *Tprogram* ⇒ *bool*

tr S mu T = (trp S mu) = 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 specifica- tions, 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 Sex, that is, we show trp Sex *µ* = Tex:

trp Sex *µ*

= [[the definition of trp]]

(trass [v0 = v0 + 7*,* v1 = 8] *µ*)@(tre (v0 + v1 ) *µ*)

= [[the definition of trass]]

(tra (v0 = v0 + 7) *µ*)@(trass [v1 = 8] *µ*)@(tre (v0 + v1 ) *µ*)

= [[the definition of tra]]

[Load *µ* (v0 )*,* Push 8*,* Add*,* Store *µ* (v0 )]@(trass [v1 = 8] *µ*)@(tre (v0 + v1 ) *µ*)

= [[the definition of *µ* : *µ* ≡ empty(v0 '→ a0)(v1 '→ a1 )]]

[Load a0 *,* Push 8*,* Add*,* Store a0 ]@(trass [v1 = 8] *µ*)@(tre (v0 + v1 ) *µ*)

= [[the definition of trass]]

[Load a0 *,* Push 8*,* Add*,* Store a0 ]@(tra (v1 = 8) *µ*)@(trass [] *µ*)

@(tre (v0 + v1 ) *µ*)

= [[the definition of tra]]

[Load a0 *,* Push 8*,* Add*,* Store a0 ]@[Push 8*,* Store *µ* (v1)]@[]@(tre (v0 + v1) *µ*)

= [[the definition of *µ* : *µ* ≡ empty(v0 '→ a0)(v1 '→ a1 )]]

[Load a0 *,* Push 8*,* Add*,* Store a0 ]@[Push 8*,* Store a1 ]@[]@(tre (v0 + v1 ) *µ*)

=

[Load a0 *,* Push 8*,* Add*,* Store a0 ]@[Push 8*,* Store a1 ]@(tre (v0 + v1 ) *µ*)

= [[the definition of tre]]

[Load a0 *,* Push 8*,* Add*,* Store a0 ]@[Push 8*,* Store a1 ]

@[Load *µ* (v0 )*,* Load *µ* (v1 )*,* Add]

= [[the definition of *µ* : *µ* ≡ empty(v0 '→ a0)(v1 '→ a1 )]]

[Load a0 *,* Push 8*,* Add*,* Store a0 ]@[Push 8*,* Store a1 ]@[Load a0 *,* Load a1 *,* Add]

=

[Load a0 *,* Push 8*,* Add*,* Store a0 *,* Push 8*,* Store a1 *,* Load a0*,* Load a1 *,* Add]

= [[the definition of Tex ]]

Tex

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

# 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 transla- tion of S. Whereas most work in this area concentrated on logics and proof techniques for such proofs, this presentation discussed the issues on how ex- isting formal specification and verification frameworks can be used as a basis for such proof generating compilers. In particular, we introduced the no- tion 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 com- piler in ML that translates a simple assignment language into a stack machine language. The mentioned flexibility was illustrated by developing two very dif- ferent, 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 ap- ply our approach to more realistic programming languages and to implement translation and proof techniques as described in [[7](#_bookmark19)] 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. *µ*Java: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbru¨ggen, 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

In this section, we present summarizations of two predefined Isabelle/HOL theories: Option and Map that we used to formalize the languages S and T . The following two sections are adapted quotations of Sections 3.4.3 and 3.4.4 from Nipkow et al. [[5](#_bookmark15)].

* 1. *Options*

datatype *α option* = None | Some *α* consts the :: *α option* → *α* primrec the (Some x) = x

the None = arbitrary

* 1. *Mappings*

types *α* ~ *β* = *α* → *β option*

empty :: *α* → *β*

empty ≡ *λ* k *.* None

( '→ ) :: (*α* ~ *β*) → *α*

→ *β* → (*α* ~ *β*)

m (x '→ y) ≡ *λ* k *.* if k = x

then (Some y) else (m k)