Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D4.6

Report on proof transforming compiler

Due date of deliverable: 2009-08-31 (T0+48)

Actual submission date: 2009-09-15

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **IoC**

Revision 8093M

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to Chapter
ETH	2, 3
IC	8
IoC	1, 4, 7, 9
IMDEA + INRIA	5, 6

This document was written by Gilles Barthe (IMDEA), Mohamed El-Zawawy (IoC), Benjamin Grégoire (INRIA), César Kunz (IMDEA), Keiko Nakata (IoC), Hermann Lehner (ETH), Peter Müller (ETH), Tarmo Uustalu (IoC), Nobuko Yoshida (IC).

Executive Summary:
Report on proof transforming compiler

This document summarises Deliverable 4.6 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr/`.

The deliverable is the final report on Task 4.4 on proof-transforming compilation. Prior work on this task was reported in Deliverables 4.3 [30] and 4.5 [31].

A proof-transforming compiler is a tool for the code producer that not only compiles a given source program, but also transforms a given proof of it (obtained, e.g., by interactive theorem proving) automatically into a proof of the compiled form. This is a lifting of the idea of a certified compiler, which automatically produces certificate of a program's conformance to some fixed safety policy, to situations where we are also interested in functional correctness or in general conformance to freely chosen policies.

The two previous deliverables treated both proof-transforming non-optimizing compilation from Java to JVML, aiming at a full-scale implementation [30], and optimizations on a more exploratory level [31].

This deliverable presents complementing work on proof and type transformation along both lines. This includes a formalized semantics of Java+JML in Coq (completing the implementation of the proof-transforming compiler for the non-optimizing case) and a case study of weakest precondition (wp) proof preservation for non-optimizing Java to JVML compilation, but also a new general Hoare proof transformability result for analysis-based optimizations, a wp proof transformation prototype for a simple optimizing compiler, a study of a big-step semantics and Hoare logic for reasoning about observable behavior of potentially nonterminating programs.

We also report on generation and preservation of linear and session types in compilation from a high-level concurrent language to binary code for distributed memory CMP.

Contents

Chapter 1

Introduction

A proof-transforming compiler is a tool for the code producer in PCC that not only compiles a given source program, but also transforms a given proof of it (obtained, e.g., by interactive theorem proving). This is a lifting of the idea of certifying or proof-generating compiler, which automatically produces of a program's conformance to some fixed safety, to situations where we concerned with functional correctness or conformance to freely chosen policies. A variation on the theme is type-generating or type-transforming compilation.

In Mobius, proof-transforming compilation was the subject of Task 4.4. Prior work on this task led to Deliverables 4.3 [30] and 4.5 [31], discussing non-optimizing compilation resp. compilation with optimizations.

In this final deliverable on the task, we complement the results of the two earlier deliverables.

Along the line of non-optimizing compilation, we complete the work on a full-scale non-optimizing Java+JML to JVML compiler by providing a Coq formalization of Java+JML and reporting a case study of proof-transforming (proof-preserving compilation) based on preservation of VCGen proof obligations in non-optimizing compilation. Concerning program optimizations, we strentghen our earlier results to cover a wide range of optimizations systematically and provide prototype implementations.

We also report on a trace-based coinductive semantics and a corresponding logic, which we intend to use in the future to obtain optimization soundness and proof transformability results accounting more adequately for nonterminating programs via observation of intermediate states. We also show how generation and preservation of linear and session types to obtain safety guarantees of compiled distributed, concurrent code.

The deliverable is based on four published papers and technical reports [26, 2, 35, 41], several rapers in the process of publication and several pieces of software, produced by the partners that contributed to the task.

The report is organized as follows. In Chapter 2, which based on [26], we report on a Coq formalization of the semantics of Java and the JML annotation language following the design of Bicolano, the Coq formalization of the JVM bytecode language semantics.

Chapter 3 presents a case study of weakest precondition proof transformation (preservation) for non-optimizing Java to JVML compilation, relying on the Coq formalization of Java+JML of Chapter 2 and Bicolano. The verification conditions generated by the VCGens for source programs and compiled code are essentially identical, so proofs for the verification conditions for a source program also prove the compiled code modulo minor modifications.

In Chapter 4, we turn to proof transformation for program optimizations and describe a generalization of our earlier results on the topic. We prove that relational program optimization soundness implies (and is in fact equivalent) to Hoare proof transformability for any analysis-based optimization described as a type system and admitting a relational interpretation of types.

Chapter 5 describes a prototype certificate translator for optimizing compilation, elaborating and implementing the method reported in [2]. It handles a class of optimizations for which assertion transformation amounts to strengthening, such as constant propagation and common subexpression elimination.

Chapter 6, based on citeNakataU09 and a paper in progress, discusses work that paves way for scaling proof-transforming compilation for finer semantics and logics. We present a novel coinductive big-step semantics for nontermination with observable intermediate states and a matching Hoare logic. The aim is to obtain a framework for proving optimizations sound and Hoare proofs transformable meaningfully also for nonterminating programs. Under finer semantics, optimization soundness is more than preserving just the return values of programs.

Chapter 7 summarizes the paper [41] by describing an approach to generation and preservation of linear and session types to accompany compilation from a high-level distributed and concurrent language through typed π-calculus to low-level code operating on distributed memory.

In the final chapter, Chapter 8, we conclude.

Chapter 2

Semantics of JML in Coq

2.1 Motivation

JML is quite popular and there are many tools that use the language for different purposes, be it a static checker, a verification tool, or a unit test generator (Place citations here). Each tool supports a different subset of JML, and, in some cases, interpret the semantics as defined in the reference manual [27] slightly differently. The reason for the ambiguity is that the reference manual is written in English, and it iss not decently possible to exactly describe a semantics of a computer language in a natural language only. We need to get help from mathematics, where it is possible to give precise and unambiguous definitions. That's were we want to go.

We are highlighting three reasons for formally defining a language like JML in a theorem prover.

A better architecture of the Proof Transforming Compiler (PTC) In the current architecture of the PTC [30], we translate the JML specifications to first order logic formulas for given program points (the annotation table). The whole verification environment then uses this FOL terms to verify a program and to produce a certificate. For a human being, it is quite hard to decide if such a formula corresponds to the original JML specifications or not. So, no matter how trustfully the verification environment is, we still need to believe that the FOL terms that we are proving against are what we specified.

One way to believe the specification is to look inside the JML to FOL translator and decide if the translation is implemented correctly—according to the reference manual. This is, as discussed already hard to decide. A better way is to get rid of the JML to FOL translator at all. Instead of using FOL terms in Coq to specify a program, we directly use JML. For this we need to embed JML in Coq. Embedding JML in Coq means to define a ADT for the language and give each construct in the ADT a meaning. In Coq, you can do this in a quite elegant way. The advantage to the JML to FOL transformer is that semantics is clearly visible and not hidden in some implementation tricks.

Eliminate uncertainty As discussed already, a reference manual in plain English leads to open questions. When defining the semantics of JML, one always needs to specify all cases and exactly describe the behavior in the mathematical model. There is no room for under-specifying the behavior of a language feature. The goal is that the formal semantics is as readable as an English text, but much more precise. We did not find a large set of inconsistencies or mistakes in the reference manual, however, many aspects are just not defined in the reference manual or just still under discussion. It is good to have a clear view on what is defined and what not.

In the formalization, it is also clear where the definition for a construct starts and where it stops. In an English text, you potentially never now if there is some more behavior or a clarification defined in another chapter because some aspects overlap.

A stable foundation for future work We hope to produce a foundation for a lot of future work that build on top of the formalized semantics of JML. Like in the case of Bicolano, we provide some hooks to the outside in order to work with the embedded JML.

2.2 Approach

In [26], we describe a formal definition of JML in the Coq proof assistant in details. Providing a formalization of JML in Coq includes:

- Definition of the domain of the language, like a heap model, numbers, or the definition of a method frame
- Definition of the syntax of Java and JML as abstract data types
- Definition of the semantics of JML constructs
- An implementation of all abstract data types mentioned above
- A translation frontend that embeds JML annotated Java programs in Coq data types
- An operational semantics for Java

In the following, we present the most important design decisions.

Build on top of Bicolano As there are many parallel aspects, we base our work on Bicolano [29] , the bytecode language definition in Coq, and adapt it in order to represent the Java source language and JML in Coq. We can keep quite some parts of Bicolano domain definition without changes, like the heap model or the definition of finite length integer numbers, but have to replace the definition of the syntax by its source code counterpart and have to write the semantics of JML constructs from scratch. Our motivation is to stay as close to Bicolano as possible, in order to enable interoperability between the two formalizations.

Only do what is really necessary We defined a 'basic' language subset that contains everything but pure syntactic sugar, and there is a lot of syntactic sugar in JML. Many constructs can be syntactically transformed into other constructs without loosing much readability, and without blowing up the specifications a lot.For this basic subset, we define the semantics in the theorem prover.

We then define the full (in the report, we call it sometimes 'extended' syntax) and provide syntactic rewriting functions within Coq. Mostly, the rewritings are pretty trivial, for instance the elimination of `non_null` in parameters by adding to all `requires` clauses the condition that the parameter is not `null`.

Using this two steps for the formalization gives us the advantage, that the semantics for JML is pretty small and readable, and many issues can already be dealt with in the syntactic rewritings from the extended syntax to the basic syntax. Of course, again, this rewritings are kept readable, in order to ensure that one can verify that the rewriting is done correctly.

Use only one state variable to represent them all A program state usually contains the current heap and frame. The frame contains the parameters and local variables, the program counter and similar information. If you use a specification language, you want to be able to refer to states other than the current state in your program. In many approaches, several state variables show up in the formalizations. For JML, this is impractical as we cannot limit the number of states that the specification has access to. It is possible to refer to a value of an expression at any labeled location in a method. That is why we store the state for all labeled instructions in a method in its frame. As we are living in Coq it is actually possible to have several copies of the whole state in a method frame.

The big advantage is that it is quite elegant to access a variable in an arbitrary (accessible) state. The semantics is clear and the signatures of our definitions stay small.

Heavily use notations in Coq One big motivation for our work is to embed the JML syntax directly in Coq so that we get rid of the cryptic FOL terms that cannot be related back to JML specifications. In order to really play that card, it is very important to have an embedding of JML in Coq that can be understood by anyone that understands JML specifications. The customer needs to be able to verify that the JML in Coq is exactly what we specified in the Java file.

That is why we spent effort in defining a large set of notations (extensions to the Coq parser) that allow us to be syntactically very close to JML. A deep embedded JML specification in Coq looks a bit strange, yet, is still understandable.

Having a form that closely resembles real JML contracts also help us writing a really trivial translation frontend that automatically generates deep embedded JML specifications for a given JML specified Java program.

Use an annotation table interface The current tool chain uses an annotation table to access the annotations for a given program point. In our formalization, we provide the same interface so that it can be integrated in the PTC with minimal effort.

2.3 Achievements

We managed to provide a formalization of a large part of JML in Coq. Our work currently covers the following aspects of the formalization.

- We defined the syntax for nearly all JML constructs that are described in the reference manual. We omitted constructs that only deals with floating point number as this is also not supported in Bicolano.
- We currently cover the semantics of JML level 0 and still add semantic definitions to more constructs.
- We provide a syntactic rewriting of the sugared JML constructs into more basic constructs and give an implementation for all JML abstract data types.
- We provide a translation frontend that embedds a given JML annotated Java program in Coq.

Chapter 3

Case study of the proof-transforming compiler

In this chapter, we report on a case study of the (non-optimizing) proof-transforming compiler (PTC). We quickly introduce the mechanism that the PTC introduces to generate bytecode certificates for source code programs and proofs. We then illustrate how proofs can be done in the PTC. We conclude the chapter with an evaluation of the proof-transforming compiler and suggestions for further improvement based on the experience from the case study.

3.1 The mechanism of the proof-transforming compiler

The PTC is described in chapter 3 and 6 of [30]. The main concept is to avoid the transformation of proofs at all. This is achieved by defining verification condition generators (VCGens) on source and bytecode level that produce the same proof obligations. The theory of preservation of proof obligations is described in [3] and chapter 4 of [30].

So, the name "proof-transforming compiler" is actually misleading as there is no (relevant) transformation of proofs involved in the process. The only differences between the verification conditions on source code and their counterpart on bytecode is the naming of variables and the absence of the boolean type on bytecode.

Figure 3.1 shows the different components of the proof-transforming compiler. In a first step, the PTC transforms the JML annotations into first order logic terms that are accessible as pre- and postconditions, as well as object invariants, local assertions and loop invariants. Bico+ generates Coq versions of the first order logic annotations and embeds the bytecode program in bicolano. The source and bytecode VCGens then use the source and the bytecode program, respectively, and the same annotations in Coq to generate equivalent (and structurally identical) verification conditions. By discharging the proof obligations on source level and with an equivalence lemma that can be proven automatically, we can prove the bytecode verification conditions.

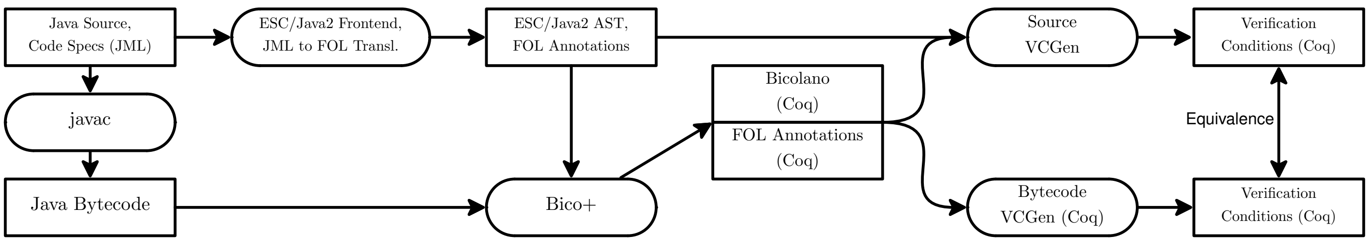
Java Source, Code Specs (JML)
ESC/Java2 Frontend, JML to FOL Transl.
ESC/Java2 AST, FOL Annotations
Source VCGen
Verification Conditions (Coq)
javac
Bicolano (Coq)
FOL Annotations (Coq)
Equivalence
Java Bytecode
Bico+
Bytecode VCGen (Coq)
Verification Conditions (Coq)

Figure 3.1: Overview of the architecture. We depict data by rectangular boxes and computations by rounded boxes.

3.2 Invoking the PTC

Since its delta release, the Mobius PVE features the proof-transforming compiler. The corresponding plugin is called *DirectVCGen*. It offers a tool bar button to generate a directory called 'mobius' in the project directory that contains all necessary files for manually proving that the program fulfills its specification:

- A background formalization containing type informations and member signatures of predefined Java Types like 'java.lang.Object', user-tactics to reduce the manual proving effort.
- A directory 'Formalisation' containing a copy of the virtual machine formalization (bicolano). This is not generated, but just extracted from an archive of the formalization that is also shipped with the Mobius PVE.
- A directory 'classes' that contains the following files for each *class*: *class*_type.v that defines the type information of the program, *class*_signatures.v that contains the signature definitions of all class members, *class*_annotations.v that contains all first order logic annotations for the given class and *class*.v that contains the coq embedded bytecode of all methods in the class.
- A directory 'vcs' that contains the generated verification conditions for the given class. There are subdirectories for each method of the class, and each subdirectory contains one or more goals to prove for the method, the source VCs, the bytecode VCs and an equivalence lemma that should be provable automatically.

3.3 Proving generated goals

We evaluated the feasibility to prove generated goals for some simple programs. In the following, we present the proof sketch for proving arithmetic properties and for properties that involve heap accesses.

3.3.1 A simple example

We use the following trivial program to illustrate the process of proving goals in the PTC. We choose two different implementations to highlight the two aspects that we present in this summary.

```
class Test {
  //@ensures \result == i * 2;
  int twice (int i) { ... }
}
```

3.3.2 First order logic annotations

Our small example contains a minimalistic JML annotation that states that for any `i`, the method `twice` returns double the value of `i`.

Although the annotation is very small, the generated first order logic formulas are considerably bigger. In the following listing, we show only the interesting (non-trivial) parts of the annotations that are generated. The specifications define the precondition and the postcondition of the method. The precondition states that `this` is a subtype of `Test` and that the invariants of all alive objects hold. The postcondition makes a case distinction on all possible constructors of `ReturnVal`. In the case of normal termination, the specification says that all invariants have to hold again and that the result is twice the input parameter. In case of exceptional termination, the specification says that no exception can be thrown (by stating `false` as goal) and that all invariants have to hold. As Coq functions are total, we also have to define the non-wellformed case `Normal None`, which stands for normal termination of the method, but without a return value, which is not possible.

```
Module twice.

 Definition mk_pre :=
  fun  (heap: Heap.t) (this: value) (i_4_23: Int.t) ⇒
    ((assign_compatible p heap this (ReferenceType (ClassType TestType.name)))
    ∧ (∀ (r0: Location) (x1:type), (((isAlive heap r0)
      ∧ (assign_compatible p heap r0 x1)) → (inv heap r0 x1)))).

 Definition mk_post :=
  fun (t:ReturnVal) (_pre_heap:Heap.t) (heap:Heap.t) (this:value) (i_4_23:Int.t) ⇒
    match t with
    | Normal (Some _result) ⇒ ((Is_true true)
      ∧ ((∀ (r1: Location) (x2:type), (((isAlive heap r1)
            ∧ (assign_compatible p heap r1 x2)) → (inv heap r1 x2)))
         ∧ ((assign_compatible p heap this (ReferenceType (ClassType TestType.name)))
            → ((vInt _result) = (Int.mul i_4_23 (Int.const (2)))))))
    | Normal None ⇒ True
    | Exception x0 ⇒
      ((((assign_compatible p heap (Ref x0) (ReferenceType
                                               (ClassType java_lang_ExceptionType.name)))
          → ((assign_compatible p heap this (ReferenceType (ClassType TestType.name)))
        → (Is_true false)))
        ∧ (∀ (r2: Location) (x3:type), (((isAlive heap r2)
           ∧ (assign_compatible p heap r2 x3)) → (inv heap r2 x3))))
      ∧ (Is_true true))
            end.
End twice.
```

3.3.3 Proving arithmetic properties

To focus on the task, we choose the following implementation for `twice`. It is not interesting to introduce local variables as those disappear in the generated proof obligations anyway.

```
//@ensures \result == i * 2;
int twice (int i) {
  return i + i;
}
```

For this method, we get one file `goal1.v` that contains one subgoal. Please note that all Coq goals in the following are simplified to only show the interesting parts. What we have to prove here, is that the precondition of `twice` implies the postcondition, where the return value is the addition of twice the input parameter. You can also see that the heap stays unchanged. In the postcondition, we feed the `heap0` to both `_pre_heap` and `heap`:

```
1 subgoal
...
```

```
______________________________________(1/1)
∀ (this0 : value) (i_3_24 : Int.t) (heap0 : Heap.t),
TestAnnotations.twice.mk_pre heap0 this0 i_3_24 →
TestAnnotations.twice.mk_post
  (Normal (Some (Num (I (Int.add i_3_24 i_3_24))))) heap0 heap0 this0 i_3_24
```

The DirectVCGen provides a set of nice user-tactics, that - in most cases - simplify the goal considerably. By default, the first proof steps are already provided:

```
nintros; repeat (split; nintros); cleanstart.
```

Applying this set of tactics lead to two subgoals:

```
2 subgoals
...
H : TestAnnotations.twice.mk_pre h1 this0 i_3_0
H0 : isAlive h1 r1 ∧ assign_compatible p h1 r1 x2
...
______________________________________(1/2)
inv h1 r1 x2



______________________________________(2/2)
Int.add i_3_0 i_3_0 = Int.mul i_3_0 2
```

The first goal origins from the part of the postcondition that states that all invariants hold in the post-state. This is simple to prove, as we didn't change the heap in the method body. We just have to extract that information form the precondition in hypothesis H

The second goal is the actual arithmetics that we are interested in. We have to prove that the addition of the same number is the same as the multiplication of that number by two in Java arithmetics with overflow. For this, we do not need any of the hypotheses that we get from the goal, but we need the formalization of numbers in bicolano. In there, we can find the data types of integer operations. We show the addition and the definition of `smod` below. `smod` is the operation that deals with overflow as specified in the Java virtual machine specifications. The addition is axiomatized by `add_prop`. The function `toZ` yields a Coq number of type `Z` for a given Java integer.

```
  Definition smod (x:Z) : Z :=
     let z := x mod base in
      match Zcompare z half_base  with
     | Lt ⇒ z
     | _ ⇒ z - base
     end.

  Parameter add : t → t → t.
  Parameter add_prop : ∀ i1 i2,
    toZ (add i1 i2)= smod (toZ i1 + toZ i2).
```

Unfortunately, this axiomatization for addition does not help us to continue the proof, as we cannot match or rewrite the term from our goal `Int.add i_3_0 i_3_0` with that axiom. Therefore, we changed the formalization of numbers in bicolano by defining the function `add` instead of axiomatizing it. In the definition below, `const` yields a Java integer number for a given Coq number, and is axiomatized for the case that the number is actually within the allowed range for Java integers. So we do not change the meaning of add anyhow, we just write it down differently.

```
Definition add (i1 : t) (i2 : t) : t := const (smod (toZ i1 + toZ i2)).
```

With this changed way of defining Java integer operations, we can now prove our goal. We begin by unfolding all arithmetic operators, which is now possible. We get the following goal:

```
Int.smod (i_3_0 + i_3_0) = Int.smod (i_3_0 * 2)
```

We now add an assertion that states the desired property for Coq numbers: `∀ i : Z ,i + i = i * 2`. We can prove this hypothesis automatically in Coq and use it to rewrite our goal so that we get the same term on both sides of the equality.

The following listing shows the full prove script of the subgoal. At some point, we tell Coq to print all coercions (implicit conversion functions from one type to another), because it helps to understand which rewritings need to be done. From the axiomatization of the function `const`, we get a new goal to prove that 2 is in the range of integer numbers. This prove can be done by actually computing the value of the type which leads to several cases, that are either trivially true or can be proven by pointing out contradictory hypotheses—we think that this kind of proofs could be further simplified by writing user tactics that deal with such basic problems.

```
    (* Goal: Int.add i_3_0 i_3_0 = Int.mul i_3_0 2 *)
    unfold Int.add, Int.mul.
    cut (∀ i : Z ,i + i = i * 2).
      intro Hcut.
      Set Printing Coercions.
      rewrite Hcut.
      rewrite Int.const_prop.
        reflexivity.
        (* Goal: Int.range 2. *)
        compute ; split ;
          [ intro Hfalse ; discriminate Hfalse
          | reflexivity ].
      (* Goal: ∀ i : Z, i + i = i * 2. *)
      auto with zarith.
```

3.3.4 Proving properties in the presence of heap changes

To focus on heap changes only, we choose the following implementation for method `twice`:

```
class Test {
  int f;
  //@ensures \result == i * 2;
  int twice (int i) {
    this.f = i*2;
    return this.f;
  }
}
```

For this, the DirectVCGen generates one file `goal1.v` that contains 6 subgoals, as many things can potentially go wrong here. In the following, we focus on 3 subgoals that are intersting.

Our specification of the postcondition says that there should be no exception. Potentially, a field update leads to a null-pointer exception and that is what see in the following goal. It's trivial to prove, as we always know that `this` cannot be `null`. So we just have to point out the contradiction between `H1` and `H4`.

```
1 subgoal
...
H1 : this0 <> Null
...
H4 : this0 = Null
...
______________________________________(1/1)
Is_true false
```

We also have to prove as another subgoal that all invariants hold in the exceptional case. Again, this is simple to prove by pointing out the contradiction in the hypotheses.

We now look at the interesting case, which says that saving `i_6_0 * 2` in `f` and then accessing this value in the new heap still yields `i_6_0 * 2`.

```
...
______________________________________(1/1)
vInt
  (Num
     (I
        (vInt
           (do_hget
              (Heap.update h1
                 (Heap.DynamicField this0 TestSignature.fFieldSignature)
                 (Num (I (Int.mul i_6_0 2))))
              (Heap.DynamicField this0 TestSignature.fFieldSignature))))) =
Int.mul i_6_0 2
```

We can rewrite the goal with the heap axiom `Heap.get_update_same` which states exactly what we need, i.e., that if we update a heap location by some value and access the same location in the new heap, we get the same value back. We get two subgoals, the first stating our property, and the second stating that accessing `f` on `this` is possible.

```
...
H: assign_compatible p h1 this0 (ReferenceType (ClassType TestType.name))
...
______________________________________(1/2)
```

```
vInt (Num (I (vInt (Num (I (Int.mul i_6_0 2)))))) = Int.mul i_6_0 2



______________________________________(2/2)
Heap.Compat h1 (Heap.DynamicField this0 TestSignature.fFieldSignature)
```

The first subgoal is can be solved automatically by Coq. The second subgoal is also very simple, but produces another three subgoals to prove, as `assign_compatible` has three inductive definitions. Two of them can be proven by contradiction (for `this0 = Null` and for the case that the receiver is an arrayi-type. So we get to the last goal to prove. We apply the right constructor of the inductive type `Heap.Compat` to exactly get `H8` as goal.

```
1 subgoal
...
cn : ClassName
...
H8 : Heap.typeof h1 loc = Some (Heap.LocationObject cn)
...
______________________________________(1/1)
Heap.Compat h1 (Heap.DynamicField loc TestSignature.fFieldSignature)
```

3.4 Conclusions from the case study

As we wanted to evaluate the proof-transforming compiler and not the user who needs to discharge the proof obligations, we have chosen to use several small examples instead of a large or sophisticated example. By this, we were able to test different aspects of the PTC without spending a large amount of time in doing proofs.

In our experiments, we encountered obstacles that forced us in some cases to go back and change small parts of the underlying theory; we mentioned the unfortunate definition of integer operations in bicolano for our purpose. Other examples are the absence of hypotheses that would help to solve a subgoal, e.g., that `this` is never `null`. Such *free* invariants of a correct Java program are missing sometimes and we added them manually to prove certain goals. However, it would be simple to add this to the verification condition generator.

After getting rid of the issues described above, we were able to prove goal generated for source code programs. However, we couldn't verify the equivalence of the source and bytecode verification conditions. We didn't manage to prove the equivalence lemma yet, neither automatically (as planned) by predefined tactics nor manually. With the information given, we couldn't spot the source of the problem.

To sum up, the proof-transforming compiler combines the advantages of doing proofs on source level with the possibility of using those proofs for bytecode certificates. Although the generated goals and annotations look awfully complicated at first, the PTC manages to present proof obligations to the customer that can be understood easily. The user-tactics that are shipped with the PTC do a very good job in simplifying the goals considerably. We could also verify that the underlying formalization (bicolano) is well-designed to prove program properties in most of the cases.

Chapter 4

Proof transformability from program transformation soundness

To obtain automatic proof transformations for program optimizations used in compilers, in prior work on Mobius (Tasks 3.5 and 4.4) we have developed a type-systematic technique of specifying analysis-guided program optimizations, enabling simple optimization soundness proofs based on relational interpretation of the type language and automatic transformation of program proofs alongside optimization [38].

We have demonstrated this method on a number of optimizations, including classical optimizations like dead code elimination, common subexpression elimination and partial redundancy elimination (PRE) for a high-level language with expressions [38, 39] and some stack usage optimizations for an operand-stack based low-level language [37]. We have also shown that type systems for analyses can be seen as applied versions of more foundational Hoare logics [18], that the approach scales for bidirectional analyses [19], and that not only program optimization, but also other automatic program transformations, e.g., analysis-guided program repair, can be treated in the same way [17].

In this chapter, we report on a generalization of the above work. In a new paper, we demonstrate that program transformation soundness implies (in fact is equivalent to) program proof transformability generally, i.e., for any program transformation specified by a type system and any notion of of validity specified by a relational interpretation of types.

Specifically, we state and prove our result in two forms.

The first takes a black-box view of the programming language, its big-step semantics and Hoare logic, the program transformation type system and the relational interpretation. We assume that the Hoare logic is sound and complete relative to the big-step semantics, but do not need to know the soundness and completeness proofs. It turns out that soundness and completeness of the Hoare logic (alongside expressibility of the semantic evaluation relation in the assertion language of the Hoare logic) are sufficient to conclude that soundness of the program transformation implies program proof transformability and vice versa.

In detail, the result is as follows.

We assume we have a big-step semantics, with evaluations of the form $\sigma \xrightarrow{s} \sigma'$, where σ, σ' are states (the pre- and poststate of a run of the statement) and a Hoare logic, with triples of the form $\{P\}s\{Q\}$, where P and Q are assertions in a signature with an extralogical constant st for the current state. We assume that the Hoare logic is sound and complete wrt. the big-step semantics in the standard sense.

We assume we are given a type system with a program transformation component where subtyping judgements are of the form $d \leq d'$ and typing judgements for statements are of the form $s : d \to d' \hookrightarrow s_*$ where d, d' are types and s, s_* are statements (an original statement and the corresponding transformed statement).

We also assume we are given, for any type d, a type-indexed relation $\sim_d$ on pairs of states (of the runs of the original and optimized statement). We assume that $\sim_d$ is expressible in the assertion language via a predicate $\cong_d$. For an assertion P and type d, we define two translations $P|_d$ and $P|^d$ by $P|_d =_{\mathrm{df}} \exists w. st \cong_d w \wedge P[w/st]$ and $P|^d =_{\mathrm{df}} \exists w. w \cong_{d'} st \wedge P[w/st]$.

We say that our program transformation is sound, if $s : d \to d' \hookrightarrow s_*$ implies that the following two conditions hold:

(i) if $\sigma \sim_d \sigma_*$ and $\sigma \xrightarrow{s} \sigma'$, then there exists σ'_* such that $\sigma' \sim_d \sigma'_*$ and $\sigma_* \xrightarrow{s} \sigma'_*$,

(ii) if $\sigma \sim_d \sigma_*$ and $\sigma_* \xrightarrow{s_*} \sigma'_*$, then there exists σ' such that $\sigma' \sim_d \sigma'_*$ and $\sigma \xrightarrow{s} \sigma'$.

We say that proofs are transformable, if $s : d \to d' \hookrightarrow s_*$ implies that the following two conditions hold:

(I) if $\{P\}s\{Q\}$, then $\{P|_d\}s_*\{Q|_{d'}\}$,

(II) if $\{P\}s_*\{Q\}$, then $\{P|^d\}s\{Q|^{d'}\}$.

It turns out that program transformation soundness is equivalent to proof transformability; in fact (i) is equivalent to (II) and (ii) to (I). The proof of uses soundness and completeness of the Hoare logic. The right-to-left implication also hinges on the expressibility of the semantic evaluation relation.

The second result is for a partially white-box setup. We can work, e.g., with the While-language of statements over some expression language. We assume the standard big-step semantics and Hoare logic rules for statements. About the program transformation type system we assume standard sequence, if, while and consequence rules (which transform the statement homomorphically); we do not restrict the assignment rules and the rules for boolean expressions.

In this setting, program transformation soundness and proof transformability are equivalent for statements, if the same holds for assignments and boolean expressions. Differently from the first, blackbox setting, the proof of this result can be conducted by direct induction on the given Hoare logic proof or semantic evaluation.

These results apply to and provide a unification of a wide range of program transformations with proof transformability. In fact, they apply to all these we had previously studied on case-by-case basis. Several variations are possible. E.g., the high-level While language can be replaced by a low-level language with jumps; the program transformation can take programs from one language to another (with a different syntax and a semantics over a different state space).

Both results are consequences of semantics/logic duality. It particular, it should be noted that not only does program transformation soundness imply proof transformability, but the converse is true as well, so the two are really two equivalent ways of stating the same property of a program transformation.

We have been developing a Coq formalization of the results.

We have also tested our approach on program analyses and optimizations we had not considered in our earlier work, most notably points-to analysis.

Chapter 5

A proof-transformation prototype

In this chapter, we report on a prototype implementation of certificate translation for a subset of the C language, and on experiments with using the prototype on small to medium size examples. This work allows us to draw for the first time some preliminary conclusions about the practicality of certificate translation. In particular, we are able to provide a preliminary analysis of the impact of certificate translation on the size of certificates, which is an essential metrics for Proof Carrying Code [36]. The tool implements a new method for transforming certificates for optimizations based on arithmetic reasoning, such as constant propagation, and common subexpression elimination. The method is simpler, in that it treats the certificate of the original program as a black-box, whereas the previous method in [2] involved weaving certificates, using a well-founded induction principle on the control flow graph of the program. In comparison, the new method also yields smaller certificates, and thus contributes to the practicality of certificate translation. The tool is available at `http://mobius.inria.fr/CertificateTranslation`.

Example 1 (Matrix Multiplication: source code) *The code given in Figure 5.1 computes the multiplication of two matrices* `a` *and* `b`*, and stores it in* `c`*. Matrices are encoded as uni-dimensional arrays; that is, if a $m \times n$ matrix A is represented by the uni-dimensional array* `a` *of size mn, we encode the array element $A_{i,j}$ as* $\mathtt{a}[i * n + j]$.

In order to support modular verification, the program verifier requires that procedures are annotated with their preconditions and postconditions, and that loops are annotated with their invariants. Procedure specifications are triples of the form $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$, where Pre and Post are assertions and annot is a partial function from program labels to assertions. Assertions are written in a language similar to (but

```
l1  :  i := 0;
l2  :  while (i < m){
l3  :     j := 0;
l4  :     while (j < p){
l5  :        k := 0;
l6  :        c[i * p + j] := 0;
l7  :        while (k < n){
l8  :           c[i * p + j] := c[i * p + j] + a[i * n + k] * b[k * p + j];
l9  :           k := k + 1;
               }
l10 :        j := j + 1;
            }
l11 :     i := i + 1;
         }
l12 :  return 0
```

Figure 5.1: Source code of matrix multiplication example

smaller than) the ACSL language used in the Frama-C project [?]; they are also allowed to refer to functions and predicates defined in an external Coq module.

Example 2 (Matrix Multiplication: specification) *The specification for matrix multiplication is given by the triple* $\langle \mathsf{Pre}, \mathsf{annot}, \mathsf{Post} \rangle$ *where:*

$$\begin{array}{ll}
\mathsf{Pre} \doteq 0 < \mathtt{m} \wedge 0 < \mathtt{n} \wedge 0 < \mathtt{p} & \\
\mathsf{annot}(l_2) \doteq \forall i, j.\ ((0 \leq i < \mathtt{i}) \Rightarrow (0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}[i,j] = (\mathtt{a} \times \mathtt{b})[i,j]) \wedge \mathtt{i} \leq \mathtt{m} & \\
\mathsf{annot}(l_4) \doteq & \forall j. ((0 \leq j < \mathtt{j}) \Rightarrow \mathtt{c}[\mathtt{i},j] = (\mathtt{a} \times \mathtt{b})[\mathtt{i},j]) \wedge (0 \leq \mathtt{j} \leq \mathtt{p}) \wedge \\
& \forall i, j.\ ((0 \leq i < \mathtt{i}) \Rightarrow (0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}[i,j] = \mathtt{c}^{l_2}[i,j]) \\
\mathsf{annot}(l_7) \doteq & \mathtt{c}[\mathtt{i},\mathtt{j}] = \sum_{r=0}^{\mathtt{k}} (\mathtt{a}[\mathtt{i},r] * \mathtt{b}[r,\mathtt{j}]) \wedge (0 \leq \mathtt{j} \leq \mathtt{p}) \wedge \\
& \forall j.\ ((0 \leq j < \mathtt{j}) \Rightarrow \mathtt{c}[\mathtt{i},j] = \mathtt{c}^{l_4}[\mathtt{i},j]) \\
\mathsf{Post} \doteq \forall i, j.\ ((0 \leq i < \mathtt{m}) \Rightarrow (0 \leq j < \mathtt{p}) \Rightarrow \mathtt{c}[i,j] = (\mathtt{a} \times \mathtt{b})[i,j])\} &
\end{array}$$

For notational convenience, we write $x[i,j]$ *instead of* $x[i * n + j]$ *if the array* x *represents an* $m \times n$ *matrix. The postcondition ensures that the array* $\mathtt{c}$ *is the result of matrix multiplication between* $\mathtt{a}$ *and* $\mathtt{b}$*, i.e. for every* i *and* j*,* $\mathtt{c}[i,j]$ *is equal to* $\sum_{k=0}^{\mathtt{n}} \mathtt{a}[i,k] * \mathtt{b}[k,j]$*, denoted* $(\mathtt{a} \times \mathtt{b})[i,j]$*. A labeled variable* $\mathtt{x}^l$ *stands for the value of the variable* $\mathtt{x}$ *at program point* l*.*

The program verifier generates for each annotated program a set of proof obligations. The generation of proof obligations proceeds in two phases: first, a symbolic execution algorithm is used to strengthen program annotations. Then, a weakest precondition calculus generates proof obligations using the strengthened annotations. In order to avoid bloated proof obligations, the weakest precondition calculus does not strengthen annotations with all the results of symbolic execution, but only with those that refer to variables that would appear in the proof obligation. The use of symbolic execution to strengthen invariants allows users provide weaker specifications.

Example 3 (Matrix Multiplication: invariant strengthening) *At program point* l_4*, a standard VC-gen would generate, for the execution that does not enter the loop, the proof obligation:* $\mathsf{annot}(l_4) \Rightarrow \neg \mathtt{j} < \mathtt{p} \Rightarrow \mathsf{annot}(l_2)[^{\mathtt{i}+1}/_{\mathtt{i}}]$*. Unfortunately, the loop invariant* $\mathsf{annot}(l_4)$ *does not provide enough information to carry the proof. In contrast, symbolic execution propagates automatically the condition provided by the outer invariants (*$\mathsf{annot}(l_2)$ *in this case) to strengthen the inner invariants, and generates provable proof obligations.*

The proof obligations are collected as lemmas in a Coq file, and must be discharged to guarantee that the program is correct w.r.t. its specification. This process is done interactively by the user, using the Coq proof assistant. The certificate of the program is the set of proof terms that are built automatically by Coq upon successful verification of the proof obligations.

Certificate translation starts after the user has completed the verification of the source program. The program is first compiled to an intermediate representation, written in a RTL language with arrays and procedure calls.

Example 4 (Matrix multiplication: RTL code) *The result of compiling the running example can be found in Figure 4. (Ignore at this point the boxes in gray, which show an optimized version of the* RTL *code).*

The checker of RTL programs is based on the same principles as the verifier for source programs: annotated RTL programs are sent to a weakest precondition calculus that generates a set of proof obligations; then the certificates are checked against the proof obligations. Unlike the verifier for source programs, the checker for RTL programs does not rely on symbolic execution—one reason for this discrepancy is that the RTL checker is trusted, and hence should be as simple as possible. To bridge the gap between the source code verifier and the RTL checker, the tool relies on a specification compiler that strengthens the original specification with the result of the symbolic execution. Although the proof obligations between source code and RTL programs are very similar, there are some minor differences that prevent directly reusing certificates—for many examples, including the running example, the proof obligations coincide syntactically. The tool implements a Coq tactic to transform the original certificates into certificates for their RTL compilation.

```
       i := 0
l2 :   set i^l2 := i
       set j^l2 := j
       set k^l2 := k
       set c^l2 := c
l2^b : cjmp i < m l2^o
       jmp l2'
l2^o : j := 0
l4 :   set j^l4 := j
       set k^l4 := k
       set c^l4 := c
l4^b : cjmp j < p l4^o
       jmp l4'
l4^o : k := 0
       r1 := i * p
       r2 := r1 + j
       c[r2] := 0
l7 :   set k^l7 := k
       set c^l7 := c
l7^b : cjmp k < n l7^o
       jmp l7'
l7^o : r3 := i * p          r3 := r1
       r4 := r3 + j         r4 := r2
       r5 := c[r4]
       r6 := i * n
       r7 := r6 + k
       r8 := a[r7]
       r9 := k * p
       r10 := r9 + j
       r11 := b[r10]
       r12 := r8 * r11
       r13 := r5 + r12
       r14 := i * p         r14 := r1
       r15 := r14 + j       r15 := r2
       c[r16] := r15
       k := k + 1
       jmp l7
l7' :  j := j + 1
       jmp l4
l4' :  i := i + 1
       jmp l2
       return 0
```

Figure 5.2: RTL representation of the matrix multiplication example

Certificate Translation for the Optimizing Phases

The tool performs a series of optimizations: constant propagation, common subexpression elimination, partial redundancy elimination, and unreachable code elimination. For each optimization, the tool transforms the program, its specification and the certificates. Previous work [2, 4] proved the existence of certificate translators for these optimizations. However, the theoretical development has not been matched by a practical implementation. As a result, it has not been possible to validate experimentally the theoretical developments, nor to assess the practicality of certificate translation. The tool described in this chapter implements an ad-hoc technique, that considers a particular class of optimizations, for which the growth of the original certificate size is moderate.

For the class of optimizations considered in this paper, the certificate translator must strengthen the specification with the result of the analysis that motivates the program transformation. Therefore, a certificate translation procedure must automatically generate a certificate for the analysis result, and then merge it with the original certificate. First, the tool defines, for each analysis module, a function that that maps every element a in A to its representation as a logical assertion. Then, from the result of the analysis as program specification, the tool computes the corresponding set of verification conditions. These proof obligations are automatically discharged in Coq by application of the ring tactic, that solves equations on ring structures by associative and commutative rewriting.

Most of the optimizations mentioned above can be classified as a replacement of expressions in the instructions of an RTL procedure, without modifying its control-flow graph. The main result of this characterization, is that the computation of verification conditions along the graph of the optimized program coincides in their logical structure with the original ones. That is, there is a syntactical correspondence between the logical formulae, up to substitution of equal expressions.

The certificate transformation implemented by the tool relies on the fact that the result of the analysis can prove the equality of the expressions in which the original and final proof obligations differ.

The tool implements a tactic that generates the new certificates, by traversing the logical structure of the proof obligations, and applying the Coq ring and rewrite tactic, taking as input a certificate for the result of the analysis.

Experimental Results

We have experimented with several examples to estimate the impact of certificate transformation in the size of the final certificates. Most of the examples are relatively small, but specifically suited to test the optimizations covered by the tool. In average, from the original certificate we have obtained a slight reduction on the certificate for the non-optimized RTL code. The size of the certificate for result of the analysis is on average 0.43 times the size of the original certificate. Merging the RTL certificates with the certificates of the analysis yields certificates that are almost three times the size of the original certificate. Certificate translation for common-subexpression elimination increases the previous certificate by a factor of 1.46 on average. In total, the final certificates are on average approximately 4 times the size of the original certificates.

We show in the following table a more detailed analysis of the certificate size for the multiplication matrix example.

PO	Source	RTL	Analysis	Merge	CSE
Pre	2922	2960	109	7615	7615
annot(l_2)	8746	8272	138	23276	23276
annot(l_4)	33232	32418	261	86962	86962
annot(l_7)	95195	93907	229	178012	253575

The table shows each certificate size for each step of the compilation. The second column represents the original certificate discharged interactively by the tool user. The third column represents the certificate size after non-optimizing compilation. The fourth column represents the size of the certificate of the analysis result, automatically generated by the tool. The fifth column represents the certificate size after merging the certificate for the RTL program and the certificate for the result of the analysis. Finally, the last column show the certificate size for the optimized program.

Other optimizations, such as redundant conditional elimination and dead code elimination, reduce the size of certificates, since verification conditions are always simplified.

Chapter 6

A trace-based coinductive big-step semantics and Hoare logic of nontermination

The standard big-step operational semantics and Hoare logic characterize only terminating runs of programs, in terms of final states or postconditions. All nonterminating runs are identified. In particular, it is not possible to see the intermediate states or output produced during such runs. At the same time, one should expect program transformations like compilation or optimizations to respect the meanings of programs on this level of detail. The challenge of stating and proving that they do so indeed calls for appropriate semantics and program logics. In this chaper, we report on work in this direction.

In [35], we propose two big-step operational semantics (relational and functional-style) for the While language, that usefully describe the behavior of potentially nonterminating programs. These semantics characterize a program run in terms of a state trace, i.e., a sequence of states that the run passes through.

In our semantics, traces are defined coinductively, as possibly infinite, nonempty sequences of states. The relational semantics is given by a statement-indexed evaluation relation between states and traces relating an initial state to corresponding trace. An auxiliary evaluation relation relates two traces: a trace already accumulated before running the statement of interest is related to a total trace after the run. The two evaluation relations are defined mutually coinductively.

In the corecursive functional semantics, the evaluation relations are replaced by evaluation functions (these are total, as any initial state leads to a trace).

A central concern in our design to is to make sure that while-loops are sufficiently productive. This is needed in order to guarantee that a nonterminating run will always be represented by an infinite trace (so that appending a further trace to it cannot grow it by further states—that would be an anomaly). We assure this by taking evaluation of an assignment or a boolean guard to constitute a single step (growing an already accumulated trace by a state). In the case of the functional semantics, this productivity is indispensable also technically: without it, the corecursive definition of the evaluation functions would not be guarded.

We prove the relational and functional big-step semantics equivalent to each other, but also to relational and functional small-step semantics, which, differently from the subtle big-step semantics case, are straightforward coinductive trace-based versions of the standard inductive state-based small-step semantics. All of these equivalence proofs are fully constructive.

We also provide a formalization of the whole development of the paper in the Coq proof assistant.

Our design was guided by the realization that nontermination (with observable intermediate states) is a monadic effect (cf. Capretta's approach to nontermination with no intermediate observation [9]). The extended evaluation function taking traces to traces is the Kleisli extension of the main evaluation function sending states to traces, a Kleisli map. The semantics of sequences is given exactly by the Kleisli composition and the semantics of while-loops hinges on the fact that the monad is completely iterative.

This work was motivated by the prospective application in certified code, but it was particularly inspired the closely related work by Leroy and Grall [28] in the framework of their CompCert certified compiler project, a major successful demonstration of the practical viability of certified compilation. They describe two attempts at big-step semantics, which we find to fall short of what one would desire in two aspects. The first semantics has separate evaluation relations for finite and infinite runs (one inductive, the other coinductive). As a result, the proof that the small-step semantics is sound relative to the big-step semantics hinges on an instance of the classical law of excluded middle, in fact, decidability of termination. Intuitively, this should not be necessary, and indeed, as our work confirms, it is just an unfortunate consequence of the chosen definition of the big-step semantics. The second has a single coinductive evaluation relation, but suffers from the anomaly that a statement following a nonterminating statement is reached and run (from an underdetermined state). In our solution, both problems are avoided.

In a newer follow-up work (still in progress), we have devised a trace-based Hoare logic to match our trace-based big-step semantics. In this logic, a triple associates to a statement a state assertion (the precondition) and a trace assertion (the postcondition). It is valid, if, for any initial state meeting the precondition, the trace produced by running the statement satisfies the postcondition. The trace assertion language is similar to that of interval temporal logics. The central connectives are chop (concatenation) and repetition (possibly infinite repetition rather than finite repetition). For both, satisfiability is defined coinductively. But provability is still inductive, i.e., derivations are wellfounded trees, as in the standard Hoare logic. This trace-based Hoare logic is sound with respective to the big-step semantics and, under the assumption of a sufficiently expressive state assertion language, also complete.

The new Hoare logic subsumes both the standard partial-correctness Hoare logic and the total-correctness Hoare logic (based on variants), there are syntactic embeddings.

Again we provide a Coq formalization of the whole development.

A curious detail that becomes apparent working with a constructive formalization in Coq is that not-nontermination and termination are distinctively different: termination is a stronger property. For example, algorithms of total unbounded search are cannot be proved terminating constructively (so intuitionistic total-correctness Hoare logic, which only detects constructive termination, can prove nothing about them). But our trace-based Hoare logic proves them constructively not-nonterminating.

We have not done so yet, but we plan to show in the closest future that the type-systematic approach to program optimizations supporting relational optimization soundness and associated automatic transformability of program proofs [38] extend to the more discriminating trace-based semantics. This means both a stronger form of optimization soundness (preservation of a finer semantics) and automatic transformation of proofs of finer program properties.

Chapter 7

Type-preserving compilations for distributed and communications programming

7.1 Motivation

In this chapter, we report a general type-preserving compilation framework from the high-level distributed or concurrent programming languages to the low-level languages based on the advanced type theory of the π-calculus. Our approach uses the typed π-calculus as an intermediate language in order to ensure a preservation of secrecy (secure information flow) and communication-safety of the high-level programming languages. Our motivations are two-fold: first it offers an effective *source* language for compilation into the low-level languages such as a typed assembly language for multicore programming. Secondly it offers an expressive *target* language into which we can efficiently and flexibly translate different kinds of high-level programs, including Web service languages [13, 23, 12, 10] and communication-based object-oriented languages [25, 14, 16]. This latter aspect is based on the observation that many concurrent and potentially concurrent programs can be represented as a collection of structured conversations, where we can abstract the structure of data movement in their programs as types for conversations. The type disciplines used in this deliverable are *linear types* and *session types* [20, 5, 8, 11, 42, 24, 15, 34, 33], and the developments of the high-level distributed languages and their types are based on the work done in Tasks 2.1 and 4.1.

7.2 Approach

We explain our framework using one example on a type-based compilation and execution framework for the low-level programming for distributed memory multicore CPUs [41]. Later we summarise how this approach is generalised and extended in Section 7.3. The basic idea is to stipulate *typed communicating processes* as a representation for an intermediate compilation step from high-level abstractions, and, after a type-based analysis of this intermediate representation, perform a *type-directed compilation* [32] onto executable binary code for distributed memory CMP. The basic step is listed in Figure 7.1(a). `L0`, `L1`, `L2`

High-level concurrent languages (`L2`)
⇓
Typed imperative processes (`L1`)
⇓
CMP executable (`L0`)

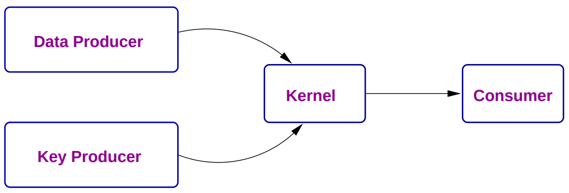

Figure 7.1: (a) The compilation framework and (b) A simple program for stream cipher

refer to abstraction levels. Each $\Downarrow$ stands for one or more type-preserving compilations. At L1, we use an intermediate concurrent imperative language with types for channel-based conversations. The preceding studies on types for communicating processes, many centring on the π-calculus, have shown that they can offer fundamental articulation and basic safety guarantee for diverse communication patterns. As communication types for the compilation framework, we use a variant of session types for multiparty interactions [8, 24, 34, 5], into which various high-level abstractions can be translated and which allows their efficient and safety-preserving compilation to distributed CMP primitives. The session types at L1 are generated from the interaction structures implicit in the high-level abstractions in L2, as we shall illustrate with a concrete example in the subsequent sections. The resulting typed communicating processes are amenable to uniform program analyses for safety assurance, and can be directly mapped to efficient code in L0, with a formal guarantee of the aforementioned key correctness properties [21, 41].

7.2.1 Streaming example and processes with session types

We take a simple program for stream cipher, depicted in Figure 7.1(b). Data Producer and KeyProducer continuously send a data stream and a key stream respectively to Kernel. Kernel calculates their XOR and sends the result to Consumer. A high-level specification of such an example can be written using a domain specific language for streaming, which we omit. Our purpose is to translate this program to a type-safe low-level program for distributed memory CMP.

We show a process representation of the streaming algorithm. In order to illustrate the key ideas, we use a simple translation scheme. In practice we use a slightly more complex, and more efficient, translation, which we shall briefly discuss at the end. The kernel initiates a session:

$$\text{Kernel} \stackrel{\text{def}}{=} \mathsf{def}\ \mathrm{K}(d,k,c)\ =\ d!\langle\rangle;\ k!\langle\rangle;\ d?(x);\ k?(y);\ c?();\ c!\langle x\ \mathsf{xor}\ y\rangle;\ \mathrm{K}\langle d,k,c\rangle\ \ \mathsf{in}\ \ \overline{a}(d,k,c).\mathrm{K}\langle d,k,c\rangle$$

Observe that the channels d and k are used for Kernel to receive data and keys from Data Producer and Key Producer, respectively. Before receiving, Kernel notifies Data/Key Producers that it is ready before receiving data/keys. Such an insertion of a notification message before the reception of datum is essential for safe translation into direct memory access operations, since without them, a datum may be written to a memory region while that region is being read and/or written by a local process, leading to inconsistent values. Note also these signals are necessary even when we use the class of streaming languages with the most regular behaviour such as static and cycle-static data flow languages. The channel c is used for Consumer to receive the encrypted data from Kernel, which is also used for notifying its readiness to receive the data. The keyword def denotes a recursive process; $\overline{a}(d,k,c)$ is a session initiation establishing a session between the three parties; $d?(x)$ is an input action at d; and $c!\langle x\ \mathsf{xor}\ y\rangle$ is an output action at c. Similarly, DataProducer and Consumer can be defined as processes with session types [21].

The exchange of messages as above forms a "conversation" among processes, with a precise structure: this structure we abstract below as a type. The session type of the Kernel is given as:

$$T_K\ =\ \mu\mathbf{t}.d!\,\langle\rangle; k!\,\langle\rangle; d?\,\langle\mathsf{bool}\rangle; k?\,\langle\mathsf{bool}\rangle; c?\,\langle\rangle; c!\,\langle\mathsf{bool}\rangle; \mathbf{t}$$

Above $\mu\mathbf{t}.T$ represents a recursive type, $k?\,\langle\mathsf{bool}\rangle$ (resp. $k!\,\langle\mathsf{bool}\rangle$) denotes the input (resp. output) of a value of bool-type, and $T;T'$ denotes a sequencing. The type of the DataProducer is given as $\mu\mathbf{t}.d?\,\langle\rangle; d!\,\langle\mathsf{bool}\rangle; \mathbf{t}$. Similarly for KeyProducer and Consumer. Safe parallel composition of communicating code is guaranteed by checking duality of types: the type of the Kernel and one of the DataProducer are dual to each other at d, so that there is no communication error occurs at d. Similarly for k and c.

7.2.2 Compiling typed processes to the low-level language

Processes with session types are guaranteed to follow rigorous communication structures, given as types. By tracing this session type, we know beforehand what and when process will send and receive as messages. Using this information, we can replace message passing in typed processes with direct memory write to a multicore chip.

```
main: {
  main: {
    r1 := getIdleCore
    r2 := getIdleCore
    r3 := getIdleCore
    r4 := getIdleCore
    fork dataProducer at r1
    fork keyProducer at r2
    fork kernel at r3
    fork consumer at r4
    yield
  }
}

dataProducer: {
  data: byte [128]
  ack: byte [0]
  main: {
    // produce data
    get ack
    put data in r3.data
    jump main
  }
}
```

```
keyProducer: {
  key: byte [128]
  ack: byte [0]
  main: {
    // produce key
    get ack
    put key in r3.key
    jump main
  }
}

consumer: {
  buf: byte [128]
  ack: byte [0]
  main: {
    get buff
    // consume buf
    put ack in r3.ack
    jump main
  }
}
```

```
kernel: {
  data: byte [128]
  key: byte [128]
  buf: byte [128]
  ackD: byte [0]
  ackK: byte [0]
  ackC: byte [0]
  main: {
    put ackD in r1.ack
    put ackK in r2.ack
    get data; get key
    r5 := 128; jump loop
   }
  loop: {
    when r5 lt 0 jump done
    r6 := data[r4]; r7 := key[r4]
    buf[r4] := r7 xor r6;
    jump loop
  }
  done: {
    get ackS
    put sum in r1.arg
    jump main
  }
}
```

Figure 7.2: L0 code for the stream example

Since our purpose is to have type-safe compilation, we use a typed assembly language, which we call L0 for brevity. L0 is built on top of MIL [40], which in turn is a multi-threaded extension of TAL [32]. Task scheduling is accomplished by loading a program into a core. This includes copying from the main memory the code and the data required for a run of the core, as well as a snapshot of the current register values.

Figure 7.2 presents one possible result of compiling our running example into L0. All typed message passing is replaced by low level primitives, using addresses of the variables in the local memory of a target core for remote asynchronous writes, where the addresses are shared at the time a thread is launched. The block associated with identifier main defines a program comprising, in this case, a single basic block, also named main. The program is intended to be uploaded at some core and its execution launched. Cores terminate their execution with a special instruction yield, thus joining the pool of available cores. Cores requiring extra workers get hold idle cores by issuing an instruction of the form r1 := getIdleCore. The first fork instruction in main.main copies program dataProducer to the core in register r1, copies a snapshot of its registers to the target core, and launches the execution of basic block dataProducer.main. Notice that by getting first the number of required cores and then forking the threads we guarantee that each thread knows all other cores (including its own) via registers r1 to r4.

The program associated with identifier dataProducer defines a program comprising two buffer declarations (named data and ack) and a basic block (named main). The core running this program writes its data buffer into the kernel's data buffer, but first needs to make sure it can overwrite the latter. Instruction get ack blocks the core until a corresponding put instruction is issued, namely via instruction put ackD in r1.ack in basic block kernel.done and the data is safely written. After put, the producer asynchronously writes its buffer (with put data in r3.data), for which the kernel waits with a get data instruction.

Program kernel declares three buffers (two incoming, one outgoing), and another three (empty) buffers used for acknowledgements, signals the data and the key producers that the respective buffers can be written, waits for the completion of the write operations, and embarks on a loop to fill the outgoing (buf) buffer. Finally it asynchronously writes this buffer into the arg buffer at core consumer before restarting the process. Instruction when r4 lt 0 **jump** done is expanded into the two instructions r4 := r4 − 1; **if** r4 == 0 **jump** done, providing for loops.

Subject reduction (Type preservation) for these languages are proved as in [24]. We can also obtain the advanced safety properties, such as communication safety (no communication mismatch) and progress (deadlock-freedom), as stated in [24, §5]. Thus the stream example in L1 is translated preserving advanced type structures for communication and distributions among the multiple participants; using our framework, the translation into the target language (L0) also preserves these properties.

7.3 Achievements

The above section described one instance which demonstrates a use of the typed π-calculus for type-preserving compilations. Extending this idea, we developed the following methods for Task 4.2, based on the work done in Task 2.1 and Task 4.1:

1. To attest the value of this type-preserving compilation framework, IC developed the type-safe and communication-safe distributed high-level languages and their typing systems. For example, Web service languages [13, 23, 12, 10] and distributed [1] and communication-based object-oriented languages [25, 14, 16].

2. To enrich the intermediate calculi, we developed the advanced type theories for the π-calculus such as the time-out language for distributions [7], multiparty session types [8, 24, 5], the types for the exceptions [11], the types for secrecy preservations [20], and higher-order mobile code [33]. In particular, the work [34] demonstrates type-preserving communication optimisations based on multiparty session types is effective to ensure the correctness and safety of a distributed algorithm for multicore programming.

3. We developed a general framework [41, 21] to compile the high-level distributed languages established in (1) into (2); then using the above method, we embed (2) into low-level languages preserving security properties such as communication-safety, deadlock-freedom and secure information flows. We also extended this framework to logics for concurrency [22, 6].

As future work, we are currently working on the experiments of the general framework proposed in this chapter via collaborations with the Sensoria project. It centres on a simple imperative concurrent language equipped with multiparty session communications and their types. The language, combined with two other associated languages, is intended to serve as an intermediate language (roughly of level L1) to which typed high-level concurrent languages such as X10 and StreamIt and others are compiled into. The details of this language and its typing system are discussed in [41]. The framework implements a series of type-directed translation steps from high-level typed concurrent languages into the low-level code targeted at the Cell architecture, using IBM's QS21 blade servers and their compiler architecture.

Chapter 8

Conclusions

We judge that the outcomes of Task 4.4 demonstrate convincingly that automatic proof transformation to accompany program compilation is viable for both functional correctness and freely chosen safety policies. This represents a significant strengthening of the original PCC scenario of certificed compilation, which means that the compiler generates proofs, but for a fixed safety policy only.

At the present stage, the Mobius PVE supports transformation of weakest precondition proofs for non-optimizing Java to JVML compilation through the Bicolano and Java+JML formalizations.

The research line on optimizing compilation had been planned as exploratory, but turned out to be highly rewarding in terms of the new knowledge obtained. It led to powerful frameworks of proving optimization soundness and extracting proof transformations from these soundness proofs in a systematic fashion. Here, full-scale implementations are future work, but we deem the first experiments to have been successful.

Bibliography

[1] A. Ahern and N. Yoshida. Formalising Java RMI with explicit code mobility. *TCS*, 389(3):341–410, 2007.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. *ACM Transactions on Programming Languages and Systems*, 31(5):18:1–18:45, June 2009.

[3] G. Barthe, B. Grégoire, and M. Pavlova. Preservation of Proof Obligations from Java to the Java Virtual Machine. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008.

[4] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *European Symposium on Programming*, volume 4960 of *Lecture Notes in Computer Science*, pages 368–382. Springer-Verlag, 2008.

[5] Andi Bejleri and Nobuko Yoshida. Synchronous multiparty session types. *Electr. Notes Theor. Comput. Sci.*, 241:3–33, 2009.

[6] M. Berger, K. Honda, and N. Yoshida. Completeness and logical full abstraction in modal logics for typed mobile processes. In *ICALP*, Lecture Notes in Computer Science, pages 99–111. Springer-Verlag, 2008.

[7] M. Berger and N. Yoshida. Timed, distributed, probabilistic, typed processes. In *APLAS*, Lecture Notes in Computer Science, pages 158–174. Springer-Verlag, 2007.

[8] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer-Verlag, 2008.

[9] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[10] M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. *ENTCS*, 171(3):127–151, 2007.

[11] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 402–417. Springer-Verlag, 2008.

[12] M. Carbone, K. Honda, and N. Yoshida. Theoretical aspects of communication-centred programming. *ENTCS*, 209:125–133, 2008.

[13] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 2007.

[14] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous session types and progress for object-oriented languages. In Marcello Bonsangue and Einar Broch Johnsen, editors, *International Conference on Formal Methods for Open Object-based Distributed Systems*, number 4468 in Lecture Notes in Computer Science, pages 1–31. Springer-Verlag, 2007.

[15] M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. On progress for structured communications. In *Trustworthy Global Computing*, Lecture Notes in Computer Science, pages 257–275. Springer-Verlag, 2008.

[16] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and session types. *Information and Computation*, 207(5):595–641, 2009.

[17] B. Fischer, A. Saabas, and T. Uustalu. Program repair as sound optimization of broken programs. In *IEEE and IFIP Int. Symp. on Theoretical Aspects of Software Engineering*, pages 165–173. IEEE Press, 2009.

[18] M. J. Frade, A. Saabas, and T. Uustalu. Foundational certification of data-flow analyses. In *IEEE and IFIP Int. Symp. on Theoretical Aspects of Software Engineering*, pages 107–116. IEEE Press, 2007.

[19] M. J. Frade, A. Saabas, and T. Uustalu. Bidirectional data-flow analyses, type-systematically. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 141–149. ACM Press, 2009.

[20] K. Honda and N. Yoshida. A uniform type structure for secure information flow. *ACM Transactions on Programming Languages and Systems*, 29(6):101 pages, 2007.

[21] Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Type-directed compilation for multicore programming. *Electr. Notes Theor. Comput. Sci.*, 241:101–111, 2009.

[22] Kohei Honda and Nobuko Yoshida. A unified theory of program logics: an approach based on the π-calculus. In *Proceeding in the International Conference, Visions for Computer Science*, pages 259–274. BCS, 2008.

[23] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Web services, mobile processes and types. *The Bulletin of the European Association for Theoretical Computer Science*, February(91):165–185, 2007.

[24] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Principles of Programming Languages*, pages 273–284. ACM Press, 2008.

[25] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *ECOOP*, Lecture Notes in Computer Science, pages 516–541. Springer-Verlag, 2008.

[26] A. Kägi, H. Lehner, and P. Müller. A formalization of JML in the Coq proof system. Technical report, ETH Zurich, 2009. Available as technical report from `http://www.pm.inf.ethz.ch/people/lehnerh/jmlcoq`.

[27] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, February 2007. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[28] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):285–305, 2009.

[29] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from `http://mobius.inria.fr`.

[30] MOBIUS Consortium. Deliverable 4.3: Intermediate report on proof-transforming compiler, 2007. Available online from `http://mobius.inria.fr`.

[31] MOBIUS Consortium. Deliverable 4.5: Report on proof transformation for optimizing compilers, 2008. Available online from `http://mobius.inria.fr`.

[32] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999. Expanded version of a paper presented at POPL 1998.

[33] D. Mostrous and N. Yoshida. Session-based communication optimisations for higher-order mobile processes. In *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 203–218. Springer-Verlag, 2009.

[34] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In G. Castagna, editor, *European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer-Verlag, 2009.

[35] K. Nakata and T. Uustalu. Trace-based coinductive semantics for While: Big-step and small-step, relational and functional styles. In T. Nipkow and C. Urban, editors, *Theorem Proving in Higher-Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 375–390. Springer-Verlag, 2009.

[36] G. C. Necula. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[37] A. Saabas and T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman and F. Spoto, editors, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 190(1) of *Electronic Notes in Theoretical Computer Science*, pages 103–119. Elsevier, 2007.

[38] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1–2):131–154, 2008.

[39] A. Saabas and T. Uustalu. Proof optimization for partial redundancy elimination. *Journal of Logic and Algebraic Programming*, 78(7):620–643, 2009.

[40] Vasco T. Vasconcelos and Francisco Martins. A multithreaded typed assembly language. In *Proceedings of TV06 - Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, pages 133–141, 2006.

[41] N. Yoshida, V. Vasconcelos, H. Paulino, and K. Honda. Session-based compilation framework for multicore programming. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science. Springer-Verlag, 2009.

[42] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93, 2007.