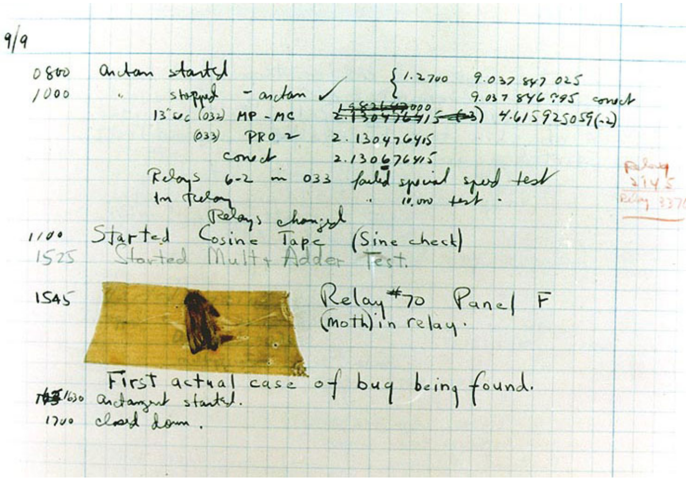

Generating Typing Proofs for Scaletta

Semester Project

10. June 2005

Author : Grégory Mermoud

Supervisor : Vincent Cremet

Professor : Martin Odersky

Contents

About the cover

The cover picture illustrates the first "computer bug" of history: a moth found trapped between points at Relay 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1945. The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program". In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia.

"To err is human, but for a real disaster you need a computer."

Chapter 1

Introduction

1.1 Goal of the project

SCALA is a functional and object-oriented language that combines both the concepts of inner classes and virtual types. An inner class C is a class nested into another one such that each instance of C contains a reference to an instance of the enclosing class. Virtual types are types whose occurence in a class needs to be reinterpreted in the context of a subclass. The combination of these two features in a type system is quite complex, but also leads to dramatically increase the expression power of the language. For instance, it enables it to encode the parametric polymorphism included in JAVA 5.

SCALETTA is an object-based calculus able to capture the essence of the SCALA type system. On one hand, the typing of this calculus is formally defined through typing rules described in Appendix A. On the other hand, the SCALETTA compiler is able to decide whether a program is well typed or not. The goal of this project is to link the formal description of the typing rules and their implementation in the compiler. More precisely, we aim to provide a formal proof of the fact that a program is well typed. We use the proof assistant COQ and the related language as target for the generated proof. Section 2.2 and 2.2.4 describes SCALETTA and COQ more in details.

Coupled with a formal proof of the fact that all well-typed programs are safe, that is not part of this project, the typing proof generation is a important step towards programs verification and certification.

In Chapter 2, we remind the concepts of inner classes and virtual types before introducing the SCALETTA calculus and the COQ proof assistant. In Chapter 3, we explain the process of translating a SCALETTA program in its COQ equivalent. Chapter 4 aims to introduce smoothly our approach to proof generation by considering only the proof that a program evaluates in a given term. In Chapter 5, we explain the main differences between generating semantic and well-formedness proofs, and we explain the main issues arising from them. Finally, Chapter 6 provides some ideas for further work and improve-

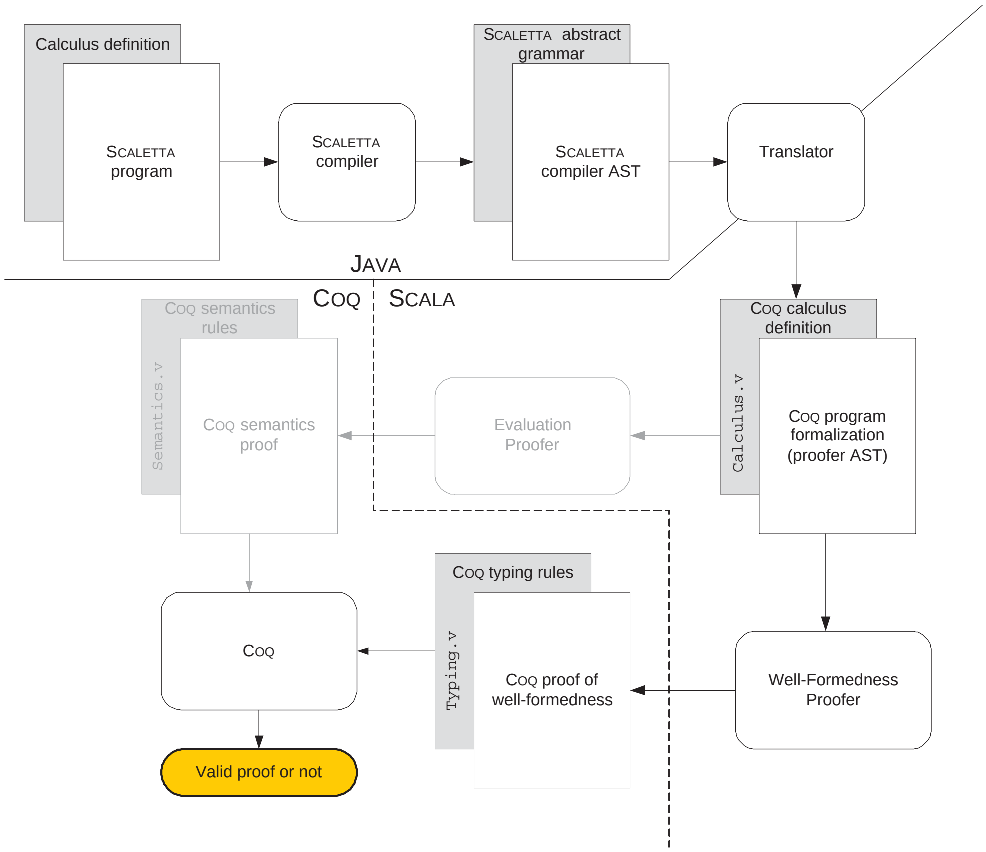

Figure 1.1: A diagram of the program flow, including the name of important files and the languages they are written in.

ments.

The Figure 1.1 provides a schematic view of the program flow, including the name of important files and the languages they are written in. You may notice that the ultimate goal of this project is to produce a COQ proof of well-formedness that can be verified by the COQ proof assistant.

Chapter 2

Background

2.1 Inner Classes and Virtual Types

This section is a summary of the technical report [AC05]. For any further details, please report to this paper, available on the SCALETTA web page [scab].

2.1.1 Inner classes

A *nested class* is a class declared within another one. We distinguish two kinds of nested classes: *inner classes* which can access the current instance of their enclosing class and *static nested classes* which can not. Within an inner class the current instance of its enclosing class is called the *current enclosing instance* and given an instance i of an inner class, it is called the *enclosing instance of* i.

Static nested classes are equivalent to top-level classes with some privileged rights to access static members of their enclosing class. Herein we are only interested in the issues posed by the presence of a current enclosing instance in inner classes.

Enclosing instances In JAVA any non-static class declared within some class `C` is an inner class `I`. Within the inner class `I`, the expression `C.this` denotes the current enclosing instance which is of type `C`.

In the code below, we consider an explicit declaration of a field `outerI` that holds the current enclosing instance.

```
public class C {
  class I { final C outerI = C.this; }
}
```

Actually, every inner class has a hidden field that holds this instance and the syntax `C.this` is simply a way to access this hidden field. We call this field *the outer field* and it is in fact the only element that differs an inner class from a static nested class.

Several issues arise from the introduction of inner classes in a language. Discussing each of them is beyond the scope of this report. You may find further details in the paper [AC05].

2.1.2 Virtual Types

In some object-oriented languages, it is possible to declare abstract type members that have no exact type value, but only a type bound. These members may then be given different type values in different subclasses and therefore their exact value depends on the exact class of the value from which they are selected. We call such type members *virtual types*. We illustrate virtual types with the following example written in SCALA that is a language that supports virtual types:

```
abstract class M {
  type T <: Object;
  val x: T;
  val y: T = x;
}
class N extends M {
  type T = String;
  val x = "foo";
}
```

In class `M`, the fields `x` and `y` are both declared with the type `T`. It is therefore legal to assign `x` to `y`. Within class M, the exact value of type `T` is unknown; it is only known that this value is bound by (is a subtype of) `Object`. Although `"foo"` has type `String` and `String` is bound by `Object`, it would be illegal to assign `"foo"` to `x` because in subclasses of `M`, `T` may be assigned any subtype, say `S`, of `Object`. If `String` is not a subtype of `S`, it would result in a typing error. Since `T` is assigned the type `String` in the subclass `N`, it is possible to assign `"foo"` to `x` in class `N`.

Mixing Virtual Types with Inner Classes

Typing virtual types in the presence of inner classes requires some kind of alias analysis, as shown in the technical report [AC05]. To illustrate this claim we consider in SCALA the following example.

```
abstract class A {
  type T <: Object;
  abstract class X {
    val outerX = A.this;
    val x: T;   // T <=> A.this.T <=> this.outerX.T
  }
}
class B extends A {
```

```
  type T = String;
  class Y extends X {
    val outerY = B.this;
    val x = "foo";
  }
}
```

In this example, it is possible to establish that the bound `this.outerX.T` of `x` resolves to the type `String` in class `Y` only because it was possible to establish that for any instance of `Y`, the fields `outerX` and `outerY` hold the same value. Without that information, one would only know that `x` has the type `T` and that `x` is bound by `Object`.

2.1.3 Summary

To sum up, typing virtual types in the presence of inner classes requires some kind of alias analysis. This analysis is needed when a virtual type has to be reinterpreted within a different context from the one where it occurs. It can be performed by reinterpreting expressions of the form `C.this`. However, it can also be performed by reinterpreting the translations of these expressions into sequences of outer field selections. The translated expressions have the advantage of being relocatable; they remain well-formed and keep the same meaning in subclasses of the class where they occur which is not the case for expressions of the form `C.this`.

2.2 Scaletta

SCALETTA is a calculus of classes and objects whose goal is to type virtual types in the presence of inner classes. The calculus has neither methods nor class constructors. Instead it has a more general concept of abstract inheritance which enables a class to extend an arbitrary object. This choice reduces greatly the number of evaluation rules.

2.2.1 Syntax

A SCALETTA program consists of a list of class declarations and a main term representing the result of the program. Each class has a name, zero or one parent, a list of field definitions, and a list of field valuations. We distinguish two types of field definition:

1. The syntax `def` $f : t$ declares a new field f which can hold a value with type bound t.

2. The syntax `typedef` $f : t$ declares a new field f which can hold a virtual type bounded by the type t.

We will see later, in Section 5, that this distinction is decisive in handling virtual typing within SCALETTA.

A field valuation `val` $f = t$ gives the value t to an already declared field f. Note that types and terms share the same syntactic structure. Within a program, classes are referred to through their name, therefore all classes must have a globally unique name.

Although all classes are declared at the top-level, all are inner classes; indeed each one has an implicit outer field and an enclosing instance has to be provided to instantiate them. To bootstrap the whole thing, there is also an implicit root class `Root` that may never be explicitly instantiated and whose unique instance is provided from the outside.

The calculus implements inheritance through delegation. This means that any instance c of class C with inherited members contains a value that implements those members. This value is called the *delegate* of c. Each time an implementation for a member is requested on c, one is first searched in class C. If none is found, the request is forwarded to the delegate of c. In our calculus, the parent of a class C is a term t that is used to compute the delegate of a new instance of class C. Note that this term is evaluated in the context of the enclosing class of C.

Terms are of four different kinds. The traditional `this` denotes the current instance. The field selection $t.f$ denotes the evaluation of the field f on the term t. he instance creation $t\texttt{!}C$ corresponds to the JAVA expression t.`new` $C()$: it creates a new instance of class C whose implicit outer field is initialized with t. The outer field selection $t\texttt{@}C$ corresponds to the JAVA expression t.`outerC` where `outerC` denotes the implicit outer field of class C. Hence it returns the content of that field, provided t is an instance of class C.

In some sense, the operation $t\texttt{@}C$ is the opposite of $t\texttt{!}C$ because it extracts from an instance of class C the enclosing instance that was used to create it. Hence it is reasonable to consider that the expression $t\texttt{!}C\texttt{@}C$ is equal to t.

2.2.2 A Basic Example

The following SCALETTA program computes the sum of two positive integers which are inductively encoded with a base class `Int` and two subclasses `Zero` and `Succ`. Please note that the code below contains a lot of syntactic sugar: for instance, the term `Int.this` is actually equivalent to `this@Succ@Int!Int`.

```
class Int {
  def pred: !Int;
  def succ: !Int = !Succ;
  def add(that: !Int): !Int;

  class Succ extends !Int {
    val pred = Int.this;                    // this@Succ@Int!Int
```

```
    val add(that) = pred.add(that.succ);
  }
}

class Zero extends !Int {
  val pred = this;
  val add(that) = that;
}

class IntStatic {
  def zero: !Int = !Zero;
  def one: !Int = zero.succ;
  def two: !Int = one.succ;
  def three: !Int = one.add(two);
}

def main: !Int = !IntStatic.three;
```

The program defines a class `Int` which contains two fields `pred` and `succ` which both hold an instance of the class `Int`, that is, the predecessor, respectively the successor of the current instance. The class `Int` also contains the method `add(that:!Int)` which takes an instance of the class `Int` as parameter and returns another `Int` instance.

The class `Succ` is defined as an inner class of `Int`. Moreover, the class `Succ` extends `Int`. Hence it has to define the value of each uninitialized field inherited from `Int`. The field `pred` holds the current enclosing instance, that is, the predecessor of this integer. Note that `Zero` has no predecessor, that is why it is not an inner class.

In class `Succ`, the method `add(that)` returns the value `pred.add(that.succ)`, that is, a recursive call to the method `add` on the predecessor of the current object. The resolution of this call will occur in the the class `Zero` whose `add` method directly returns its parameter. Thus, we can expect the value `one.add(two)` to be resolved as `this!Zero!Succ!Succ!Succ`, that is, the SCALETTA term encoding the value 3.

2.2.3 The Scaletta compiler

SCALETTA is not a strictly theoretical calculus since there is a compiler and a interpreter for it. The compiler transforms a SCALETTA program in a abstract syntax tree (AST). It also desugarizes the code (methods, blocks with nested definitions, anonymous classes), performs a name analysis and checks the typing. This compiler is written in JAVA and it uses a JAVA extension supporting algebraic types, PICO [pic], to define the AST.

Our project goal is to re-use the AST provided by the compiler in order to generate two different proofs for each SCALETTA program P: (1) P evaluates in a term t and (2) P is well-formed (or its typing is correct). To validate the proofs, we are going to use Coq, a computer-aided proof assistant.

2.2.4 The Coq Proof Assistant

Developed in the LogiCal project, the Coq tool is a formal proof management system: a proof done with Coq is mechanically checked by the machine. In particular, Coq allows:

- the definition of functions or predicates,
- to state mathematical theorems and software specifications,
- to develop interactively formal proofs of these theorems,
- to check these proofs by a small certification *kernel*.

Coq is based on a logical framework called "Calculus of Inductive Constructions" extended by a modular development system for theories. Basically, it accepts two sort of proofs: (1) tactic-based proofs which are closer to the human way of proving and (2) exact proofs that are terms of the calculus.

Chapter 3

Translation

In this phase, we re-use the AST provided by the original SCALETTA compiler and we translate it into another AST that is closer to the COQ syntax. Note that, for the project, we start from an AST where all names have been resolved, which simplifies the writing of our proof generators. This makes a great difference because in the SCALETTA compiler name analysis is intrinsically complex since it has to be performed simultaneously with type-checking. For instance to relate the application of a method f to its definition, it is necessary to first determine a type T for the receiver object of the application. Only then is it possible to lookup the name f in the type T.

Since COQ is a functional calculus, we decide to use a functional language such as SCALA to implement our proof generator. Thus, we need to translate a JAVA-PICO data structure, the SCALETTA compiler AST, into a SCALA one. Fortunately, this can be easily achieved thanks to the great interoperability of SCALA with JAVA. Indeed, both languages share the same foundations[1].

3.1 The Scaletta compiler AST

The main nodes of the SCALETTA compiler AST are class symbols which contain lists of references to the super classes[2], the inner classes, the fields and their values, if any.

The SCALETTA fields are also represented by symbols which store the type of the field and a boolean value that is true if the field defines conceptually a type or a value. The types are represented by the class `CType` which provides bounds. In the scope of this project, a field has only one bound, defined as a SCALETTA term. Moreover, each field can have a valuation that is represented by a map that associates a term to a field in the AST.

[1]For more information on the relations between SCALA and JAVA, please visit the SCALA web page [scaa].

[2]In this version of SCALETTA, one can define only one super class, but the compiler was designed in order to easily support future extensions.

The SCALETTA terms are encoded by case classes which form a small subtree within the AST. On the top level, a term can be either a `This` or a selection. Among the selection case, there is three different selectors: the field selection $t.f$, the *outer* field selection t@C and the instance creation $t!C$.

A SCALETTA program is encoded by a class symbol that represents the class `Root`. From this point, every class of the program is accessible through recursive calls on the list of inner classes.

3.2 The proofer AST

The AST of the SCALETTA compiler presents two drawbacks:

- It is not close enough to the COQ representation. Therefore, it does not allow an elegant translation of the program to a COQ syntax.
- It is written in JAVA and we want to write our proofer in SCALA in order to benefit of the fact that it is a functional language, like COQ.

Therefore, we need to translate the compiler AST into a SCALA AST that will be used for further phases of the project. The SCALA code below is a slightly simplified version of the actual code.

```
// Scala data structure to represent a Scaletta program in Coq
class CoqTree;

// Program node
case class CoqProg (
  olabels       : Map[String,OLabel],
  clabels       : Map[String,CLabel],
  flabels       : Map[String,FLabel],
  getClasz      : Map[CLabel, mkClass],
  getField      : Map[FLabel, mkField],
  getFieldValue : Map[Pair[CLabel, FLabel], Option[Term]],
  getMain       : Term
) extends CoqTree;

// Labels (Classes, Fields, Owners)
case class CLabel(name: String) extends CoqTree;

case class FLabel(name: String, typedef: boolean) extends CoqTree;

abstract class OLabel extends CoqTree;
  case class Root extends OLabel;
  case class OClasz(L: CLabel) extends OLabel;

// Term node
abstract class Term extends CoqTree;
```

```
  // This
  case class This extends Term;

  // Selectors
  // Instance creation -> t!L
  case class New(t: Term, L: CLabel) extends Term;

  // Field selector -> t.l
  case class Get(t: Term, l: FLabel) extends Term;

  // Outer field selector -> t@L
  case class Out(t: Term, L: CLabel) extends Term;

// Field node
abstract class Field extends CoqTree;
  case class mkField(L: CLabel, t: Term) extends Field;

// Class node
abstract class Clasz extends CoqTree;
  case class mkClass(owner: OLabel, t: Option[Term]) extends Clasz;
```

This data structure is almost an exact traduction of the calculus definition in Coq (Calculus.v, Appendix B). Its main interest is the fact that it uses case classes which allows us to perform pattern matching.

3.3 Translation

The first step consists of running the original Scaletta compiler on the program to be translated. Then, we use the resulting AST. We can easily identify the main term since this is the last valued field of the `Root` class. Then, we recursively visit all its nodes twice:

1. The first run collects all labels and builds the lists `olabels`, `clabels` and `flabels`.
2. The second runs builds the remaining lists such as `getClasz`, `getField` and `getFieldValue`.

The final step is dedicated to print the proofer AST by using the Coq syntax. This can be done very easily because of the great similarity of the proofer AST and the actual calculus definition. The code below illustrates the translation of the example of Section 2.2.2, that computes the sum of two positive integers.

```
Require Calculus.

Module MyProgram.
```

```
(** Class Label **)
Inductive MyCLabel : Set :=
 | id_6_Succ      : MyCLabel
 | id_3_IntStatic : MyCLabel
 | id_19_0_add    : MyCLabel
 | id_11_0        : MyCLabel
 | id_21_0        : MyCLabel
 | id_2_Zero      : MyCLabel
 | id_1_Int       : MyCLabel
 | id_5_add       : MyCLabel
 | id_10_0_add    : MyCLabel
.


(** Field Label **)
Inductive MyFLabel : Set :=
 | id_8_succ   : MyFLabel
 | id_7_pred   : MyFLabel
 | id_12_zero  : MyFLabel
 | id_17_o_    : MyFLabel
 | id_15_three : MyFLabel
 | id_14_two   : MyFLabel
 | id_13_one   : MyFLabel
 | id_16_i_0   : MyFLabel
 | id_9_add    : MyFLabel
.

Definition CLabel: Set := MyCLabel.
Definition FLabel: Set := MyFLabel.

Definition CLabelDec: forall (L M: CLabel), {L = M} + {L <> M}.
Proof. decide equality. Qed.
Definition FLabelDec: forall (l m: FLabel), {l = m} + {l <> m}.
Proof. decide equality. Qed.

(** Class owner labels - O P Q **)
Inductive OLabel  : Set :=
| root          : OLabel
| class         : CLabel -> OLabel.
(** Terms - p q t u v w x y z **)
Inductive Term    : Set :=
  | this          : Term
  | new           : Term -> CLabel -> Term
  | get           : Term -> FLabel -> Term
  | out           : Term -> CLabel -> Term.
(** Field definitions **)
Inductive Field   : Set :=
```

```
  | mkField       : CLabel (** Field owner **)
                -> Term   (** Field bound **)
                -> Field.
(** Class definitions **)
 Inductive Class   : Set :=
  | mkClass       : OLabel      (** Class owner **)
                -> option Term (** Class super **)
                -> Class.

Definition getClass(L: CLabel): Class :=
  match L with
   | id_11_0   => (mkClass (class id_3_IntStatic) (Some (get (get
       this id_13_one) id_9_add)))
   | id_2_Zero         => (mkClass root (Some (new this id_1_Int)))
   | id_19_0_add       => (mkClass (class id_6_Succ) (Some (new this
       id_5_add)))
   | id_1_Int  => (mkClass root None)
   | id_10_0_add       => (mkClass (class id_2_Zero) (Some (new this
       id_5_add)))
   | id_6_Succ         => (mkClass (class id_1_Int) (Some (new (out
       this id_1_Int) id_1_Int)))
   | id_3_IntStatic    => (mkClass root None)
   | id_21_0   => (mkClass (class id_19_0_add) (Some (get (get (out
       this id_19_0_add) id_7_pred) id_9_add)))
   | id_5_add  => (mkClass (class id_1_Int) None)
  end.

Definition getField(l: FLabel): Field :=
  match l with
     | id_8_succ       => (mkField id_1_Int (new (out this id_1_Int)
         id_1_Int))
     | id_17_o_        => (mkField id_5_add (new (out (out this
         id_5_add) id_1_Int) id_1_Int))
     | id_13_one       => (mkField id_3_IntStatic (new (out this
         id_3_IntStatic) id_1_Int))
     | id_12_zero      => (mkField id_3_IntStatic (new (out this
         id_3_IntStatic) id_1_Int))
     | id_15_three     => (mkField id_3_IntStatic (new (out this
         id_3_IntStatic) id_1_Int))
     | id_9_add        => (mkField id_1_Int (new this id_5_add))
     | id_14_two       => (mkField id_3_IntStatic (new (out this
         id_3_IntStatic) id_1_Int))
     | id_16_i_0       => (mkField id_5_add (new (out (out this
         id_5_add) id_1_Int) id_1_Int))
     | id_7_pred       => (mkField id_1_Int (new (out this id_1_Int)
         id_1_Int))
  end.

Definition getFieldValue(L: CLabel)(m: FLabel): option Term :=
```

```
    match L,m with
       | id_19_0_add , id_17_o_  => (Some (get (new this id_21_0)
           id_17_o_))
       | id_3_IntStatic , id_15_three    => (Some (get (new this
           id_11_0) id_17_o_))
       | id_2_Zero , id_7_pred   => (Some this)
       | id_2_Zero , id_9_add    => (Some (new this id_10_0_add))
       | id_11_0 , id_16_i_0     => (Some (get (out this id_11_0)
           id_14_two))
       | id_6_Succ , id_7_pred   => (Some (out this id_6_Succ))
       | id_10_0_add , id_17_o_  => (Some (get this id_16_i_0))
       | id_3_IntStatic , id_12_zero     => (Some (new (out this
           id_3_IntStatic) id_2_Zero))
       | id_3_IntStatic , id_13_one      => (Some (get (get this
           id_12_zero) id_8_succ))
       | id_1_Int , id_8_succ    => (Some (new this id_6_Succ))
       | id_21_0 , id_16_i_0     => (Some (get (get (out this id_21_0)
           id_16_i_0) id_8_succ))
       | id_6_Succ , id_9_add    => (Some (new this id_19_0_add))
       | id_3_IntStatic , id_14_two      => (Some (get (get this
           id_13_one) id_8_succ))
       | _ , _ => None
    end.

  Definition getMain: Term :=
    (get (new this id_3_IntStatic) id_15_three)
    .
End MyProgram.
```

Chapter 4

Evaluation

The main purpose of this phase is to generate a proof of the fact that a program, say P, evalutes in a term, say t. In fact, such a proof is not really useful and we implement this part rather as a practice before going on some harder stuff than as an actual step toward our final goal. Indeed, semantics and typing rules are very similar and such an approach is really valuable since the last part of the project was far easier after this starter.

4.1 From Semantic Rules to Implementation

The program semantics is defined by a set of rules which are divided into two categories: (1) reduction and (2) expansion. The whole SCALETTA semantics is available in Appendix A. Herein we will show how a given rule is actually implemented in the proofer.

Let us consider the following rule which claims that a term $t_1 = t\texttt{!}L$ can be expanded to a term t_2, that is, the result of substituting t for `this` in the declared parent of class L.

$$(\prec -Ext) \quad \frac{\text{getClassSuper}(L) = u}{t\texttt{!}L \prec \{t/\texttt{this}\}u} \tag{4.1}$$

The following code shows how the $(\prec -Ext)$ rule is translated in a inductive COQ rule called `Exp_Ext`. You may find the whole semantics in COQ in Appendix B.

```
Inductive Exp: Term -> Term -> Prop :=
 | Exp_Refl  :
   forall (t: Term),
     (Exp t t)
 | Exp_Trans :
   ...
 | Exp_Ext   :
```

```
  forall (t u: Term) (L: CLabel) (O: OLabel),
   (getClass L = mkClass O (Some u)) ->
   (Exp (new t L) (append t u))
| Exp_Red   :
  ...
```

All relations are inductively defined on inductive sets. Hence, their implementation is naturally based on recursive functions, selecting the right branch by using pattern matching. Considering the expansion for example, we perform pattern matching on the term t in order to decide which rule to apply. Thus, if t is an instance of the case class `New(t,L)`, we apply the constructor `Exp_Ext`; otherwise, we apply another constructor of `Exp`. When several cases are possible, we have to define strategies. This can be done without restricting the generality of the formalization, since the rules are designed to be confluent in the sense that choosing a rule instead of another one never leads to a dead end, provided they are both applicable in this case.

4.1.1 Evaluating

The proofer implementation in Scala includes recursive functions of the form:

$$\begin{aligned} f: \quad & \mathbb{T} \mapsto (\mathbb{T}, \mathbb{P}) \\ & t \rightarrow (u, p) \end{aligned} \tag{4.2}$$

where $\mathbb{T}$, $\mathbb{P}$ are respectively the set of terms and evaluation proofs, and t, u p are respectively a term, its evaluation and the proof of the fact that t evaluates to u.

The proofer is roughly composed of 5 functions:

1. The function `isValue` performs pattern matching on its parameter t and returns `true` if t is a value, `false` otherwise. This function is also recursive: for instance, if t is of the form v**!**C, we returns the result of `isValue(`v`)`.

2. The function `evaluate` that takes as parameter a term t and returns both its evaluation and the proof of it. It is a typical recursive function that is splitted into two cases `Step` and `End`. The former first reduces the term t in a term u and then calls again `evaluate` on u ; the latter directly returns t, provided that t is a value. The choice among these cases is done by calling the boolean function `isValue` on t.

3. The function `append` takes two parameters t, u and appends t to u, i.e. it substitutes t for `This` in u.

4. The function `red` takes as parameter a term t and returns its one-step reduction. This function includes many strategies in order to choose the right constructor to apply. Further details are provided in Section 4.1.2.

5. The function `exp` involves a slightly more complex approach. Indeed, we do not want to find any expansion of a term t, but one satisfying a given condition. Hence the function `exp` returns the one-step expansion of its parameter, but we seldom directly call this function. Instead, we use another function `lookupExp` that recursively calls `exp` until the obtained expansion satisfies a condition defined as a function $f_c : \mathbb{T} \mapsto \mathbb{B}$, where $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$.

4.1.2 Strategic choices

Since some rules are ambiguous, we need to choose which constructor to apply. Instead of picking up one at random, we prefer to set up strategies. Herein, we give further details on the strategies involved by the function `red`. The above SCALA code is the skeleton of `red`.

```
def red(t: Term): Pair[Term, Red] = t match  {
  case New(t1,label) =>     // Red_CNew

  case Get(t1,label) =>     // Strategy needed
    isValue(t1) match {
      case true  =>         // Red_Get
      case false =>         // Red_CGet
    }
  case Out(t1,label) =>     // Strategy needed
    isValue(t1) match {
      case true  =>         // Red_Out
      case false =>         // Red_COut
    }
  case _ =>
  }
```

In the case that t is of the form $t_1.f$ or $t_1 @ C$, a strategy is needed because two rules are applicable. Here we simply test whether t_1, the prefix of t, is a value or not. If it is, we apply the rule `Red_Get`, respectively `Red_Out`. Otherwise, we apply the rule `Red_CGet`, respectively `Red_COut`. This strategy implements the simple idea of giving priority to contextual rules to reduce the prefix of a term. Once the prefix is a value, we can apply the central rules of the semantics, namely `Red_Get`, `Red_Out` and `Exp_Ext`. This makes sense because a value is anyway not reducible.

4.2 Proving an evaluation

Up to this section, we have not explained how we can produce proofs. In fact, the evaluation process is only a mean of proving that a program (or rather its main term) evaluates to a given term.

4.2.1 Data Structure for Proofs

The first step toward this goal is to define a data structure for these proofs, that is very close to the actual Coq structure. The code below provides a snippet of both data structures.

In Coq:

```
Inductive Red: Term -> Term -> Prop :=
 | Red_CNew:
   forall (t u: Term) (L: CLabel),
     (Red t u) -> (Red (new t L) (new u L))
 | Red_CGet:
   forall (t u: Term) (l:F FLabel),
     (Red t u) -> (Red (get t l) (get u l))
```

In Scala:

```
trait ProofTerm;

abstract class Red extends ProofTerm;
case class RedCNew(t: Term, u: Term, L: CLabel, H: Red)
  extends Red;
case class RedCGet(t: Term, u: Term, l: FLabel, H: Red)
  extends Red;
```

This representation allows to perform pattern matching on each proof term. It is particularly interesting when implementing the proofer since it enables the compiler to detect many errors which would not have been detected otherwise. For instance, if we are waiting for a proof term `Red` and we get `Exp`, then this error will arise at compile time. Moreover, the greatest advantage of this representation is that it is easy to build in parallel with the evaluation process and easy to print in the Coq format.

4.2.2 Proof Generation

A Coq proof can be either an exact term of the underlying calculus or a flow of tactics. We choose the former rather than the latter because our approach is far better suited to a calculus term, naturally inductive, than to a flow of tactics, naturally iterative. The main drawback is the fact that a human cannot read the generated proofs even if they are very simple. The proof term below intends to illustrate this fact; it is a snipet of the proof that the example of Section 2.2.2 whose goal is to compute the sum of 1 and 2 evaluates in a term that actually corresponds to 3.

```
Require Semantics.
Module MySemantics := Semantics.SetProgram(MyProgram).
Import MyProgram.
Import MySemantics.
```

```
Lemma value: MySemantics.EvaluateMain (new (new (new (new this
id_2_Zero) id_6_Succ) id_6_Succ) id_6_Succ).


exact(Evaluate_Step (get (new this id_3_IntStatic) id_15_three) (get
(new (new this id_3_IntStatic) id_11_0) id_17_o_) (new (new (new (new
this id_2_Zero) id_6_Succ) id_6_Succ) id_6_Succ) (Red_Get (new this
id_3_IntStatic) this (get (new this id_11_0) id_17_o_) id_15_three
id_3_IntStatic (Exp_Refl (new this id_3_IntStatic)) (refl_equal (Some
(get (new this id_11_0) id_17_o_)))) (Evaluate_Step (get (new (new
this id_3_IntStatic) id_11_0) id_17_o_) (get (new (new (new this
id_3_IntStatic) id_11_0) id_21_0) id_17_o_) (new (new (new (new this
id_2_Zero) id_6_Succ) id_6_Succ) id_6_Succ) (Red_Get (new (new this
id_3_IntStatic) id_11_0) (new (new this id_2_Zero) id_6_Succ) (get
(new this id_21_0) id_17_o_) id_17_o_ id_19_0_add (Exp_Trans (new (new
this id_3_IntStatic) id_11_0) (get (get (new this id_3_IntStatic)
id_13_one) id_9_add) (new (new (new this id_2_Zero) id_6_Succ)

...

id_1_Int) id_1_Int))))) (Exp_Red (new (out (new this id_2_Zero)
id_1_Int) id_1_Int) (new this id_1_Int) (Red_CNew (out (new this
id_2_Zero) id_1_Int) this id_1_Int (Red_Out (new this id_2_Zero) this
id_1_Int (Exp_Ext (new this id_2_Zero) (new this id_1_Int) id_2_Zero
root (refl_equal (mkClass root (Some (new this id_1_Int))))))))))))
(refl_equal (Some (new this id_6_Succ)))) (Evaluate_End (new (new (new
(new this id_2_Zero) id_6_Succ) id_6_Succ) id_6_Succ) (IsValue_Red
(new (new (new this id_2_Zero) id_6_Succ) id_6_Succ) id_6_Succ
(IsValue_Red (new (new this id_2_Zero) id_6_Succ) id_6_Succ
(IsValue_Red (new this id_2_Zero) id_6_Succ (IsValue_Red this
id_2_Zero (IsValue_Value))))))))))))))))))).
Qed.
```

The process of generating such a proof from our dedicated data structure is straightforward since it consists in printing each node recursively. The main issue arises from filling the data structure.

Building a Proof

Recall that each method of the proofer returns a term u resulting from the evaluation, the expansion or the reduction of the parameter t, together with a proof of that fact. Basically, a proof of an evaluation, say e, is a term that has the same structure as the tree of calls which lead to e.

Actually, it works because the evaluation is designed to fit to a proof generation and relies on the semantics rules. It shows that these rules are expressive enough to evaluate any Scaletta program.

Chapter 5

Well-Formedness

This phase is the last one and its purpose is to type-check a SCALETTA program and to generate a proof of the fact that this program is actually well-formed. We say that a program P is well-formed if and only if it respects the type system rules.

As you will notice, this phase is quite similar to the previous one from many points of view. That is why we are not going to provide highly detailed explanation, but only the points the two phases differ in.

5.1 From Typing Rules to Implementation

The typing rules have roughly the same structure as the semantic rules, but they differ in two important points.

The first difference is that for the semantics, all terms occurring through the evaluation process were interpreted in the context of the implicit root instance, as the main term of the program from which they stem. For typing, terms must be interpreted in the context of their enclosing class. To denote the interpretation of a term t in the context of a class L, we just replace in t the initial `this` with the abstract root $[L]$.

The other difference is that a type-checker must deal with abstract fields, i.e. fields whose value is unknown in the current context. Fortunately there exists a way of compensating this lack of information, it consists in approximating them with their declared bound.

To show the general form of a typing rule, we just comment the following expansion rule.

```
Inductive Exp: ATerm -> ATerm -> Prop :=
 | Exp_This:
   forall (C: CLabel) (o: OLabel) (s: option Term),
     (getClass C = mkClass o s) ->
     (Exp (This (class C)) (New (This o) C))
```

This rule claims that a term of the form $[C]$, written `(This (class C))` in Coq, can be expanded to $[o]!C$, written `(New (This` o`)` C`)` in Coq, where o corresponds to the enclosing class of C. In other words, in the context of a class C, the current instance `this` is known to be an instance of C. Furthermore, its C enclosing instance is known to be the current instance of the enclosing class of C.

You may get a complete list of typing rules in Appendix B.

5.1.1 Type Fields and Term Fields

For this phase, we modified the original compiler in order to add a field property which specifies whether a field denotes a type or a term (a value). This piece of information is really important for defining typing strategies: contrary to type fields, the exact value of a term field is never used for type-checking; a term field can only be approximated by its declared bound.

In the Scaletta code below the class `A` defines two fields: `T`, that denotes a type, and `x`, that denotes a value. In the original compiler, the type of a field is inferred by the context, but we cannot easily do the same. Hence we slightly modified the Scaletta syntax in order to introduce a new key word that is `typedef` instead of the inexpressive `def`.

```
class A {
  typedef T: !Object;
  def x: T;
}

class B extends !A {
  val T = !Int;
  val x = 3;
}
```

If `T` was declared as a term field (using `def` instead of `typedef`), the `x` valuation would not be well-formed in class `B`, because `T` would only be known by its bound, i.e. `!Object` and `3` is not an instance of all subtypes of `Object` (for instance `3` is not an instance of `List`).

But declaring `T` as a type field allows the type-checker to make use of its value in class `B`, i.e. `!Int`, in order to accept the `x` valuation.

The rule of `WF_Valuation` formalizes the idea that the bound of a field must be re-interpreted in every class that contains a valuation for this field.

```
Inductive WF_Valuation: CLabel -> FLabel -> Term -> Prop :=
 | WF_Val:
   forall (L: CLabel) (l: FLabel) (t u: Term) (M: CLabel) (u1: ATerm),
     (getField l) = (mkField M u) ->           (** def 1 **)
     (WF_Term (append (This (class L)) t)) ->  (** hyp 1 **)
     (Inst (This (class L)) (class M)) ->      (** hyp 2 **)
     (Red (append (This (class L)) u) u1) ->   (** hyp 3 **)
```

```
(Exp (append (This (class L)) t) u1) ->   (** hyp 4 **)
(WF_Valuation L l t).
```

The rule claims that a field valuation `val` $l = t$ is well-formed in the context of a class L if and only if the following properties are satisfied:

`(hyp 1)` The term t is well-formed in the context of the class L.

`(hyp 2)` The current instance of L is an instance of the class M, the owner of the field l.

`(hyp 3)` The bound u of the field l can be reduced to a term u_1 in the context of the class L.

`(hyp 4)` The term t can be expanded to u_1 in the context of L.

Let us recall the above example for illustrating the rule `WF_Valuation`. In this case, the term t is 3, the field l is `x`, the class L is B and the owner M of the field `x` is A. Hence we can verify that it is actually well-formed by checking each hypothesis. The first two are quite straightforward, but some difficulties arise with the others.

5.1.2 Proving `WF_Valuation`

The main problem with `WF_Valuation` is that we need to find a term u_1 which satisfies both the third and the fourth hypothesis. Therefore, we face an alternative, that is, either (1) we first reduce completely the bound u and then we look for an expansion of t that matches or (2) we perform an interlinked search. The latter can provide a complete set of possible values for u_1 and some shorter proofs, but the former is conceptually far simpler and therefore easier to implement.

Again, our goal is not to produce short proofs which are easily readable by humans. Hence, we prefer a simple and elegant solution to another one leading to the same result in practice, we choose the first strategy.

5.1.3 The Lemmas

To conclude the proof that a program is well-formed we need three lemmas: `proveGetSuper`, `proveGetField` and `proveGetFieldValue`. Their goal is to simplify the proof by factorizing it. Indeed, the term `WF_Prog`, that is the proof that a program is well-formed, is composed of 4 sub-proofs which claim that the main term, each field valuation, each field bound and each super class is well-formed. The first one is not a problem since the main term is unique, but the others have to be generated for each field valuation, bound and super class.

That is why we use lemmas as shortcut. The Coq code below provides an illustration of the lemma `proveGetFieldValue`, which relies on another lemma `impliesGetFieldValue` that is fixed and therefore not provided herein.

```
Lemma proveGetFieldValue:
 forall (P: CLabel -> FLabel -> Term -> Prop),
  ...
  (P Li li ti) ->
  ...

  (forall (L: CLabel) (l: FLabel) (t: Term),
    (getFieldValue L l) = (Some t) ->
    (P L l t)).
Proof.
  intros.
  apply GetFieldValue_ind; trivial.
  apply impliesGetFieldValue; trivial.
Qed.
```

Basically, we have to fill this skeleton with a list of proofs that each field valuation is well-formed. Instead of `P Li li ti`, we write `WF_Valuation L l t` where `L, l` and `t` are respectively a class label, a field label and a term among all the field valuations of a given program.

Thus, the generation is quite straightforward since we simply iterate on the lists `getClass`, `getField` and `getFieldValue` that constitutes the Scala representation of a Scaletta program.

5.2 Main Differences with Semantics

Even if the principles remain the same as in the semantics, many parts are more complex, such as the proof of `WF_Valuation` detailled in Section 5.1.2. Most of the problems arise from the fact that even a reduction may fail in the type checker; for instance the selection $t.l$ of an abstract type l cannot be reduced even though it is not a value. In `WF_Valuation` we look for the most reduced term u_1 of a term u, i.e. we define u_1 as the reduction of u such as any further reduction would fail. Hence, we have to handle the case when it actually fails. This can be done by using the `Option` class in Scala.

Chapter 6

Conclusion

The original goal and the main contribution of this project is to prove the expressiveness of the SCALETTA calculus by generating a proof of well-formedness for a large and meaningful SCALETTA program. This program computes prime numbers based on the sieve of Eratosthenes implemented by lists of integers. More precisely it contains the definitions of booleans, integers and lists of integers. These inductive datatypes are modeled as classes using the visitor design pattern. Visitors are polymorphic in their result type and polymorphism is naturally encoded via virtual types.

Moreover, it allowed to discover some errors in the typing rules by implementing them thoroughly. We are now absolutely sure that they are coherent while allowing the implementation of expressive programs.

This report and the source code of the program can be useful for further projects in the field of typing proofers. The next step is to make this program support the new features of further versions of SCALETTA and deal efficiently with even larger programs such as SCALA programs translated in SCALETTA.

Nevertheless, to fulfill these requirements, it would be necessary to attach more importance to its optimization. For example, it would be interesting to keep track of the analysis of a term in order to avoid analyzing another occurrence of the same term.

Appendix A

Scaletta formalization

In Figure A.1, we provide the rules which formalize both the syntax and the semantics of SCALETTA. Figure A.2 contains rules for the abstract reduction and expansion, as well as the formal definition of a well-formed term. Figure A.3 provides a formal definition of a well-formed program.

These rules are written in mathematical notation. You can find their COQ version in Appendix B.

$$
\begin{array}{llcll}
\text{Class name} & L, M & & & \\
\text{Field name} & l, m & & & \\
\text{Term} & t, u, v & ::= & \texttt{this} & \text{current instance} \\
 & & \mid & t\,\texttt{!}L & \text{instance creation} \\
 & & \mid & t\texttt{@}L & \text{outer field selection} \\
 & & \mid & t\,.\,l & \text{field selection} \\
\text{Owner} & O & ::= & \texttt{Root} & \text{root} \\
 & & \mid & L & \text{class} \\
\text{Value} & V & ::= & \texttt{this} & \\
 & & \mid & V\,\texttt{!}L &
\end{array}
$$

$$
\begin{array}{r|lcl}
 & \text{getClassOwner}(L) & = & O \\
\text{partial} & \text{getClassSuper}(L) & = & t \\
 & & & \\
 & \text{getFieldOwner}(l) & = & L \\
 & \text{getFieldBound}(l) & = & t \\
\text{partial} & \text{getFieldValue}(L, l) & = & t \\
 & & & \\
 & \text{getMain} & = & t
\end{array}
$$

$(t \rightarrow u)$ **Reduction**

$$(\rightarrow\text{-cnew}) \quad \frac{t \rightarrow u}{t\,\texttt{!}L \rightarrow u\,\texttt{!}L}$$

$$(\rightarrow\text{-cout}) \quad \frac{t \rightarrow u}{t\texttt{@}L \rightarrow u\texttt{@}L}$$

$$(\rightarrow\text{-cget}) \quad \frac{t \rightarrow u}{t\,.\,l \rightarrow u\,.\,l}$$

$$(\rightarrow\text{-out}) \quad \frac{t \prec u\,\texttt{!}L}{t\texttt{@}L \rightarrow u}$$

$$(\rightarrow\text{-get}) \quad \frac{t \prec u\,\texttt{!}L \qquad \text{getFieldValue}(L, l) = v}{t\,.\,l \rightarrow \{t/\texttt{this}\}v}$$

$(t \prec u)$ **Expansion**

$$(\prec\text{-red}) \quad \frac{t \rightarrow u}{t \prec u}$$

$$(\prec\text{-refl}) \quad \frac{}{t \prec t}$$

$$(\prec\text{-trans}) \quad \frac{t \prec u \qquad u \prec v}{t \prec v}$$

$$(\prec\text{-super}) \quad \frac{\text{getClassSuper}(L) = u}{t\,\texttt{!}L \prec \{t/\texttt{this}\}u}$$

Figure A.1: Syntax and Semantics

$$\text{Rooted term} \quad \hat{t}, \hat{u}, \hat{v} ::= [O] \mid \hat{t}\,\texttt{!}L \mid \hat{t}\,\texttt{@}L \mid \hat{t}\,.\,l$$

Abstract Reduction $(\hat{t} =: \hat{u})$

(=:-refl) $$\frac{}{\hat{t} =: \hat{t}}$$

(=:-trans) $$\frac{\hat{t} =: \hat{u} \qquad \hat{u} =: \hat{v}}{\hat{t} =: \hat{v}}$$

(=:-cnew) $$\frac{\hat{t} =: \hat{u}}{\hat{t}\,\texttt{!}L =: \hat{u}\,\texttt{!}L}$$

(=:-cout) $$\frac{\hat{t} =: \hat{u}}{\hat{t}\,\texttt{@}L =: \hat{u}\,\texttt{@}L}$$

(=:-cget) $$\frac{\hat{t} =: \hat{u}}{\hat{t}\,.\,l =: \hat{u}\,.\,l}$$

(=:-out) $$\frac{\hat{t} <: \hat{u}\,\texttt{!}L}{\hat{t}\,\texttt{@}L =: \hat{u}}$$

(=:-get) $$\frac{\hat{t} <: \hat{u}\,\texttt{!}L \qquad \text{getFieldValue}(L, l) = v}{\hat{t}\,.\,l =: \{\hat{t}/\texttt{this}\}v}$$

Abstract Expansion $(\hat{t} <: \hat{u})$

(<:-red) $$\frac{\hat{t} =: \hat{u}}{\hat{t} <: \hat{u}}$$

(<:-refl) $$\frac{}{\hat{t} <: \hat{t}}$$

(<:-trans) $$\frac{\hat{t} <: \hat{u} \qquad \hat{u} <: \hat{v}}{\hat{t} <: \hat{v}}$$

(<:-super) $$\frac{\text{getClassSuper}(L) = u}{\hat{t}\,\texttt{!}L <: \{\hat{t}/\texttt{this}\}u}$$

(<:-this) $$\frac{\text{getClassOwner}(L) = O}{[L] <: [O]\,\texttt{!}L}$$

(<:-get) $$\frac{\text{getFieldBound}(l) = u}{\hat{t}\,.\,l <: \{\hat{t}/\texttt{this}\}u}$$

Instance Test $(\hat{t} : O)$

(:-root) $$\frac{\hat{t} <: [\texttt{Root}]}{\hat{t} : \texttt{Root}}$$

(:-class) $$\frac{\hat{t} <: \hat{u}\,\texttt{!}L}{\hat{t} : L}$$

Rooted Term Well-formedness $(\hat{t}\,\diamond)$

(⋄-this) $$\frac{}{[O]\,\diamond}$$

(⋄-out) $$\frac{\hat{t}\,\diamond \qquad \hat{t} : L}{\hat{t}\,\texttt{@}L\,\diamond}$$

(⋄-new) $$\frac{\hat{t}\,\diamond \qquad \hat{t} : \text{getClassOwner}(L)}{\hat{t}\,\texttt{!}L\,\diamond}$$

(⋄-get) $$\frac{\hat{t}\,\diamond \qquad \hat{t} : \text{getFieldOwner}(l)}{\hat{t}\,.\,l\,\diamond}$$

Figure A.2: Typing 1/2

Program Well-formedness

$$(\diamond\text{-super})\quad \frac{\{[\text{getClassOwner}(L)]/\texttt{this}\}t\ \diamond}{\text{getClassSuper}(L) = t\ \diamond}$$

$$(\diamond\text{-bound})\quad \frac{\{[\text{getFieldOwner}(l)]/\texttt{this}\}t\ \diamond}{\text{getFieldBound}(l) = t\ \diamond}$$

$$(\diamond\text{-main})\quad \frac{\{[\texttt{Root}]/\texttt{this}\}t\ \diamond}{\text{getMain} = t\ \diamond}$$

$$(\diamond\text{-val})\quad \frac{\begin{array}{c}\hat{t} = \{[L]/\texttt{this}\}t \\ \hat{t}\ \diamond \\ [L] : \text{getFieldOwner}(l) \\ \{[L]/\texttt{this}\}\text{getFieldBound}(l) =: \hat{u} \\ \hat{t} <: \hat{u}\end{array}}{\text{getFieldValue}(L, l) = t}$$

Figure A.3: Typing 2/2

Appendix B

Scaletta formalization in Coq

B.1 Scaletta Calculus

```
(** Program definitions **)
Module Type Program.

  (** Class labels - K L M **)
  Parameter CLabel    : Set.
  Parameter CLabelDec: forall (L M: CLabel), {L = M} + {L <>
     M}.

  (** Field labels - k l m **)
  Parameter FLabel    : Set.
  Parameter FLabelDec: forall (l m: FLabel), {l = m} + {l <>
     m}.

  (** Class owner labels - O P Q **)
  Inductive OLabel   : Set :=
    | root           : OLabel
    | class          : CLabel -> OLabel.

  (** Terms - p q t u v w x y z **)
  Inductive Term     : Set :=
    | this           : Term
    | new            : Term -> CLabel -> Term
    | get            : Term -> FLabel -> Term
    | out            : Term -> CLabel -> Term.

  (** Field definitions **)
  Inductive Field    : Set :=
    | mkField        : CLabel (** Field owner **)
                    -> Term    (** Field bound **)
                    -> Field.

  (** Class definitions **)
```

```
  Inductive Class    : Set :=
    | mkClass        : OLabel      (** Class owner **)
                    -> option Term (** Class super **)
                    -> Class.

  (** Returns the definition of a class **)
  Parameter getClass: CLabel -> Class.

  (** Returns the definition of a field **)
  Parameter getField: FLabel -> Field.

  (** Returns the valuation of a field in a class **)
  Parameter getFieldValue: CLabel -> FLabel -> option Term.

  (** Returns the main term. **)
  Parameter getMain: Term.

End Program.
```

Listing B.1: Calculus.v

B.2 Semantics Rules

```
Require Import Calculus.

Module SetProgram(MyProgram: Program).

Import MyProgram.

(** Substitution of t for this in u **)
Fixpoint append (t u: Term) {struct u}: Term :=
  match u with
    | this       => t
    | (new u1 L) => (new (append t u1) L)
    | (out u1 L) => (out (append t u1) L)
    | (get u1 l) => (get (append t u1) l)
  end.

(** Expansion **)
Inductive Exp: Term -> Term -> Prop :=

  | Exp_Refl:
    forall (t: Term),
      (Exp t t)
```

```
  | Exp_Trans:
    forall (t u v: Term),
      (Exp t u) -> (Exp u v) -> (Exp t v)

  | Exp_Ext:
    forall (t u: Term) (L: CLabel) (O: OLabel),
      (getClass L = mkClass O (Some u)) ->
      (Exp (new t L) (append t u))

  | Exp_Red:
    forall (t u: Term),
      (Red t u) -> (Exp t u)

(** Reduction **)
with Red: Term -> Term -> Prop :=

  | Red_CNew:
    forall (t u: Term) (L: CLabel),
      (Red t u) -> (Red (new t L) (new u L))

  | Red_CGet:
    forall (t u: Term) (l: FLabel),
      (Red t u) -> (Red (get t l) (get u l))

  | Red_COut:
    forall (t u: Term) (L: CLabel),
      (Red t u) -> (Red (out t L) (out u L))

  | Red_Out:
    forall (t u: Term) (L: CLabel),
      (Exp t (new u L)) -> (Red (out t L) u)

  | Red_Get:
    forall (t u x: Term) (l: FLabel) (L: CLabel),
      (Exp t (new u L)) ->
      (getFieldValue L l = Some x) ->
      (Red (get t l) (append t x)).

(** Values **)
Inductive IsValue: Term -> Prop :=
  | IsValue_Value: (IsValue this)
  | IsValue_Red:
    forall (v: Term) (L: CLabel),
      (IsValue v) -> (IsValue (new v L)).

(** Evaluation **)
Inductive Evaluate: Term -> Term -> Prop :=
  | Evaluate_End:
    forall (t: Term),
```

```
      (IsValue t) -> (Evaluate t t)
  | Evaluate_Step:
    forall (t u v: Term),
      (Red t u) ->
      (Evaluate u v) -> (Evaluate t v).

Lemma Evaluation: forall (t u: Term), (Evaluate t u) -> (
   IsValue u).
Proof.
intros.
induction H.
trivial. trivial.
Qed.

Definition EvaluateMain (t: Term): Prop :=
  Evaluate getMain t.

End SetProgram.
```

Listing B.2: Semantics.v

B.3 Typing Rules

```
Require Import Calculus.

Module SetProgram(MyProgram: Program).

Import MyProgram.

(** Abstract Terms - p q t u v w x y z **)
Inductive ATerm     : Set :=
  | This            : OLabel -> ATerm
  | New             : ATerm -> CLabel -> ATerm
  | Get             : ATerm -> FLabel -> ATerm
  | Out             : ATerm -> CLabel -> ATerm.

(** Substitution of t for this in u **)
Fixpoint append (t: ATerm) (u: Term) {struct u}: ATerm :=
  match u with
    | this       => t
    | (new u1 L) => (New (append t u1) L)
    | (out u1 L) => (Out (append t u1) L)
    | (get u1 l) => (Get (append t u1) l)
  end.
```

```
(** Abstract Expansion **)
Inductive Exp: ATerm -> ATerm -> Prop :=

  | Exp_Refl:
    forall (t: ATerm),
      (Exp t t)

  | Exp_Trans:
    forall (t u v: ATerm),
      (Exp t u) -> (Exp u v) -> (Exp t v)

  | Exp_Ext:
    forall (t: ATerm) (u: Term) (L: CLabel) (o: OLabel),
      (getClass L = mkClass o (Some u)) ->
      (Exp (New t L) (append t u))

  | Exp_Red:
    forall (t u: ATerm),
      (Red t u) -> (Exp t u)

  | Exp_This:
    forall (L: CLabel) (o: OLabel) (s: option Term),
      (getClass L) = (mkClass o s) ->
      (Exp (This (class L)) (New (This o) L))

  | Exp_Def:
    forall (t: ATerm) (l: FLabel) (L: CLabel) (u: Term),
      (getField l) = (mkField L u) ->
      (Exp (Get t l) (append t u))

(** Abstract Reduction **)
with Red: ATerm -> ATerm -> Prop :=

  | Red_CNew:
    forall (t u: ATerm) (L: CLabel),
      (Red t u) -> (Red (New t L) (New u L))

  | Red_CGet:
    forall (t u: ATerm) (l: FLabel),
      (Red t u) -> (Red (Get t l) (Get u l))

  | Red_COut:
    forall (t u: ATerm) (L: CLabel),
      (Red t u) -> (Red (Out t L) (Out u L))

  | Red_Out:
    forall (t u: ATerm) (L: CLabel),
      (Exp t (New u L)) -> (Red (Out t L) u)
```

```
  | Red_Get:
    forall (t: ATerm) (u: Term) (l: FLabel) (L: CLabel),
      (Inst t (class L)) ->
      (getFieldValue L l = Some u) ->
      (Red (Get t l) (append t u))

  | Red_Refl:
    forall (t: ATerm),
      (Red t t)

  | Red_Trans:
    forall (t u v: ATerm),
      (Red t u) -> (Red u v) -> (Red t v)

(** Instance **)
with Inst: ATerm -> OLabel -> Prop :=

  | Inst_Root:
    forall (t: ATerm),
       (Exp t (This root)) ->
       (Inst t root)

  | Inst_Class:
    forall (t u: ATerm) (L: CLabel),
      (Exp t (New u L)) ->
      (Inst t (class L)).

(** Abstract term Well-formedness **)
Inductive WF_Term: ATerm -> Prop :=

  | WF_This:
    forall (o: OLabel),
      (WF_Term (This o))

  | WF_New:
    forall (t: ATerm) (L: CLabel) (o: OLabel) (s: option Term
       ),
      (WF_Term t) ->
      (getClass L) = (mkClass o s) ->
      (Inst t o) ->
      (WF_Term (New t L))

  | WF_Out:
    forall (t: ATerm) (L: CLabel),
      (WF_Term t) ->
      (Inst t (class L)) ->
      (WF_Term (Out t L))

  | WF_Get:
```

```
    forall (t: ATerm) (l: FLabel) (L: CLabel) (u: Term),
      (WF_Term t) ->
      (getField l) = (mkField L u) ->
      (Inst t (class L)) ->
      (WF_Term (Get t l)).

Inductive WF_Valuation: CLabel -> FLabel -> Term -> Prop :=

  | WF_Val:
    forall (L: CLabel) (l: FLabel) (t u: Term) (M: CLabel) (
        u1: ATerm),
      (getField l) = (mkField M u) ->
      (WF_Term (append (This (class L)) t)) ->
      (Inst (This (class L)) (class M)) ->
      (Red (append (This (class L)) u)  u1) ->
      (Exp (append (This (class L)) t)  u1) ->
      (WF_Valuation L l t).

Inductive WF_Program: Prop :=

  | WF_Prog:

    (forall (L: CLabel) (o: OLabel) (t: Term),
      (getClass L) = (mkClass o (Some t)) ->
      (WF_Term (append (This o) t))) ->

    (forall (l: FLabel) (L: CLabel) (t: Term),
      (getField l) = (mkField L t) ->
      (WF_Term (append (This (class L)) t))) ->

    (forall (L: CLabel) (l: FLabel) (t: Term),
      (getFieldValue L l) = (Some t) ->
      (WF_Valuation L l t)) ->

    (WF_Term (append (This root) getMain)) ->

    WF_Program.

End SetProgram.
```

Listing B.3: Typing.v

Appendix C

Main sources

C.1 Proof term data structure

```
package scaletta.linked.full.proofer;

import scaletta.linked.full.translator.CoqTree;
import scaletta.linked.full.translator.Term;
import scaletta.linked.full.translator.CLabel;
import scaletta.linked.full.translator.FLabel;
import scaletta.linked.full.translator.OLabel;
import scaletta.linked.full.translator.mkClass;

trait ProofTerm;

abstract class Equ extends ProofTerm;

case class EquClass(L: CLabel, t: mkClass) extends Equ;
case class EquField(C: CLabel, f: FLabel, t: Term)  extends
   Equ;

// Inductive Expansion
abstract class Exp extends ProofTerm with Red;

case class ExpRefl(t: Term) extends Exp;
case class ExpTrans(t: Term, u: Term, v: Term, H1: Exp, H2:
   Exp) extends Exp;
case class ExpExt(t: Term, u: Term, l: CLabel, O:OLabel, H:
   Equ) extends Exp;
case class ExpRed(t: Term, u: Term, H: Red) extends Exp;

// Reduction
abstract class Red extends ProofTerm;

case class RedCNew(t: Term, u: Term, L: CLabel, H: Red)
   extends Red;
```

```
case class RedCGet(t: Term, u: Term, l: FLabel, H: Red)
   extends Red;
case class RedCOut(t: Term, u: Term, L: CLabel, H: Red)
   extends Red;
case class RedOut(t: Term, u: Term, L: CLabel, H: Exp)
   extends Red;
case class RedGet(t: Term, u: Term, x: Term, l: FLabel, L:
   CLabel, H1: Exp, H2: Equ) extends Red;

// Values
abstract class IsValue extends ProofTerm;

case class IsValueValue(t : Term) extends IsValue;
case class IsValueRed(v: Term, L: CLabel, H: IsValue) extends
    IsValue;

// Evaluation
abstract class Evaluate extends ProofTerm;

case class EvaluateEnd(t: Term, H: IsValue) extends Evaluate;
case class EvaluateStep(t: Term, u: Term, v: Term, H1: Red,
   H2: Evaluate) extends Evaluate;
```

Listing C.1: ProofTerm.scala

C.2 Semantics Proofer

```
package scaletta.linked.full.proofer;

import scaletta.linked.full.translator.CoqTree;
import scaletta.linked.full.translator.Term;
import scaletta.linked.full.translator.CoqProg;
import scaletta.linked.full.translator.mkClass;
import scaletta.linked.full.translator.This;
import scaletta.linked.full.translator.New;
import scaletta.linked.full.translator.Get;
import scaletta.linked.full.translator.Out;


class Proofer {

  def proof(prog: CoqProg): Pair[Term,ProofTerm] = {
    evaluate(prog,prog.getMain);
  }
```

```
def isValue(prog: CoqProg, t: Term): Pair[Boolean, IsValue]
    = t match {
  case This() => // IsValue_Value
    new Pair(true, new IsValueValue(t));
  case New(v,label) => // IsValue_Red
    val Pair(b,h) = isValue(prog,v);
    new Pair(b, new IsValueRed(v,label,h));
  case _ => new Pair(false,null);
}

def append(prog: CoqProg, t: Term, u: Term): Term = u match
    {
  case This() => t;
  case New(u1,label) => new New(append(prog, t,u1),label);
  case Out(u1,label) => new Out(append(prog, t,u1),label);
  case Get(u1,label) => new Get(append(prog, t,u1),label);
  case _ => System.out.println("Error: unkown (probably
      null) term in append!"); null;
}

def evaluate(prog: CoqProg, t: Term): Pair[Term, Evaluate]
    = {
  val isval = isValue(prog, t);
  if (isval._1) {
    new Pair(t,new EvaluateEnd(t, isval._2));
  } else {
    val Pair(u,h1) = red(prog, t);
    val Pair(v,h2) = evaluate(prog, u); // (Red t u) -> (
        Evaluate u v) -> (Evaluate t v)
    new Pair(v, new EvaluateStep(t, u, v, h1, h2));
  }
}

def red(prog: CoqProg, t: Term): Pair[Term, Red] = t match
    {
  case New(t1,label) => // Red_CNew
    val Pair(u,h) = red(prog,t1);
    val res = new New(u,label);
    new Pair(res, new RedCNew(t1,u,label,h));
  case Get(t1,label) => // Red_CGet
    val Pair(isval,isval_proof) = isValue(prog,t1);
    if (!isval) { // strategy choice Red_CGet
 val Pair(u,h) = red(prog,t1);
 val res = new Get(u,label);
 new Pair(res, new RedCGet(t1,u,label,h)); // (Red t u) ->
     (Red (get t l) (get t u))
    } else { // Red_Get
 val Pair(lu,hlu) = lookup(prog, t1, x: Term => x match {
```

```
        // (Exp t (new u L)) -> (getFieldValue L l = Some u
            )
    case New(u, labelnew) if (isValue(prog,u)._1) => prog.
        getFieldValue.contains(new Pair(labelnew,label)) && !
        prog.getFieldValue(new Pair(labelnew,label)).isEmpty;
    case _ => false;
  });
  val u = lu.asInstanceOf[New].t;
  val labelnew =  lu.asInstanceOf[New].L;
  val Some(x) = prog.getFieldValue(Pair(labelnew, label));
  val res = append(prog,t1,x);
  new Pair(res, new RedGet(t1, u, x, label, labelnew, hlu,
      new EquField(labelnew, label, x))); // (Exp t (new u L)
      ) -> (getFieldValue L l = Some u) -> (Red (get t l) (
      append t u))
      }
   case Out(t1,label) => // Red_COut
      val Pair(isval,isval_proof) = isValue(prog,t1);
      if (!isval) { //strategy choice: (IsValue t) ->
          Red_COut
  val Pair(u,h) = red(prog,t1);
  val res = new Out(u,label);
  new Pair(res, new RedCOut(t1,u,label,h));
      } else { // Red_Out
  val lu = lookup(prog, t1, x: Term => x match {
    case New(u,labelnew) if (isValue(prog,u)._1) => label ==
          labelnew;
    case _ => false;
  });
  val u = lu._1.asInstanceOf[New];
  new Pair(u.t, new RedOut(t1, u.t, u.L, lu._2));
      }
   case _ => System.out.println("Error: unknown term > " + t
       ); null;
}


/* Lookup in expanded terms set for a term t matching with
    f */
def lookup(prog: CoqProg, t: Term, f: Term => Boolean):
    Pair[Term,Exp] = {
   if (f(t)) {
     new Pair(t, new ExpRefl(t)); // Exp_Refl
   } else {
      val stepexp = exp(prog,t); // one step expansion
      if (f(stepexp._1)) {
  stepexp;
      } else { // Exp_Trans
  val morestep = lookup(prog, stepexp._1, f);
```

```
    new Pair(morestep._1, new ExpTrans(t, stepexp._1, morestep
        ._1, stepexp._2, morestep._2));
        }
      }
    }

    // One-step expansion of term t
    def exp(prog: CoqProg, t: Term): Pair[Term, Exp] = t match
      {
      case New(t1,label) if (isValue(prog,t1)._1) =>
        val mkc : mkClass = prog.getClasz.apply(label).
            asInstanceOf[mkClass];
        val u = mkc.t.get;
        val res = append(prog, t1, u);
        new Pair(res, new ExpExt(t, u, label, mkc.owner, new
            EquClass(label,mkc)));
      case _ => // Exp_Red
        val Pair(tred,h) = red(prog, t);
        new Pair(tred, new ExpRed(t, tred, h));
    }
}
```

Listing C.2: Proofer.scala

C.3 Well-Formedness Proofer

```
package scaletta.linked.full.typechecker;

import scaletta.linked.full.translator.CoqTree;
import scaletta.linked.full.translator.Term;
import scaletta.linked.full.translator.CoqProg;
import scaletta.linked.full.translator.Clasz;
import scaletta.linked.full.translator.mkClass;
import scaletta.linked.full.translator.This;
import scaletta.linked.full.translator.New;
import scaletta.linked.full.translator.Get;
import scaletta.linked.full.translator.Out;
import scaletta.linked.full.translator.Field;
import scaletta.linked.full.translator.mkField;
import scaletta.linked.full.translator.OLabel;
import scaletta.linked.full.translator.CLabel;
import scaletta.linked.full.translator.FLabel;
import scaletta.linked.full.translator.Root;
import scaletta.linked.full.translator.OClasz;
```

```
class WFProofer {

  def proof(prog: CoqProg): WFProofTerm = {

    def append(t: ATerm, u: Term): ATerm = u match {
      case This() => t;
      case New(u1,label) => ANew(append(t,u1), label);
      case Out(u1,label) => AOut(append(t,u1), label);
      case Get(u1,flab) => AGet(append(t,u1), flab);
      case _ => error("TC - Error: unkown (probably null)
          term in append!");
    }


    def lookupExp(t: ATerm, f: ATerm => Boolean): Option[Pair
        [ATerm, Exp]] = {
      if (f(t)) {
   Some(Pair(t, ExpRefl(t))); // Exp_Refl
      } else {
   exp(t) match  { // one step expansion
     case None => None;
     case Some(Pair(stepexp, stepexph)) =>
       if (f(stepexp)) {
         Some(Pair(stepexp, stepexph));
       } else { // Exp_Trans
         lookupExp(stepexp, f) match {
      case Some(Pair(morestep, moresteph)) => (Some(Pair(
          morestep, ExpTrans(t, stepexp, morestep, stepexph,
          moresteph))): Option[Pair[ATerm, Exp]]);
      case None => None;
         }
       }
   }
      }
    }

    // one-step expansion
    def exp(t: ATerm): Option[Pair[ATerm, Exp]] = {
      red(t) match {
   case None => // all but Exp_Red
     t match {
       case ANew(t1,label) =>  // Exp_Ext
       val mkc = prog.getClasz.apply(label).asInstanceOf[
           mkClass];
       mkc.t match {
         case Some(u) => Some(Pair(append(t1, u), ExpExt(t1,
             u, label, mkc.owner, EquClass(label,mkc))));
         case None => None;
       }
```

```
      case AThis(Root()) => None;
      case AThis(OClasz(label)) =>
        val mkc = prog.getClasz.apply(label).asInstanceOf[
            mkClass];
      val res = ANew(AThis(mkc.owner), label);
      Some(Pair(res, ExpThis(label, mkc.owner, mkc.t,
          EquClass(label,mkc))));

      case AGet(t1, flabel) => // Exp_Def
        val mkf = prog.getField.apply(flabel).asInstanceOf[
            mkField];
      val res = append(t1,mkf.t);
      Some(Pair(res, ExpDef(t1,flabel,mkf.L,mkf.t, EquField(
          mkf.L,flabel,mkf.t))));
      case _ => None; // reduction failed and not in above
          cases
    }
  case Some(Pair(tred, redProof)) => // ExpRed
    Some(Pair(tred, ExpRed(t, tred, redProof)));
     }
   }

   // lookup for reduction
   def lookupRed(t: ATerm, f: ATerm => Boolean): Option[Pair
       [ATerm, Red]] = {
     if (f(t)) {
  Some(Pair(t, RedRefl(t))) // Exp_Refl
     } else {
  red(t) match  { // one step expansion
    case None => None;
    case Some(Pair(stepred, stepredh)) =>
      if (f(stepred)) {
        Some(Pair(stepred, stepredh))
      } else { // Exp_Trans
        lookupRed(stepred, f) match {
     case Some(Pair(morestep, moresteph)) => (Some(Pair(
         morestep, RedTrans(t, stepred, morestep, stepredh,
         moresteph))): Option[Pair[ATerm, Red]])
     case None => None
         }
      }
  }
     }
   }


   // one-step reduction
   def red(t: ATerm): Option[Pair[ATerm, Red]] = {
```

```
    t match {
  case ANew(t1,label) => // Red_CNew
    red(t1) match {
      case Some(Pair(u,h)) =>
        Some(Pair(ANew(u,label), RedCNew(t1,u,label,h)))
      case None => None
    }

  case AGet(t1,flab) => // Red_Get or Red_CGet ?
    red(t1) match {
      case None if flab.typedef => // strategy choice
          Red_Get if t1 is irreductible and label is a
          typedef
        lookupExp(t1,{
     case ANew(tnew, newlab) if prog.getFieldValue.contains(
         Pair(newlab, flab)) => prog.getFieldValue(Pair(
         newlab, flab)) match {
       case Some(nimp) => true
       case None => false
     }
     case _ => false
        }) match {
     case Some(Pair(ANew(tnew, newlab), exph)) => prog.
         getFieldValue(Pair(newlab, flab)) match {
       case Some(u) => Some(Pair(append(t1,u), RedGet(t1,u,
           flab,newlab, InstClass(t1, tnew, newlab, exph),
           EquFieldValue(newlab , flab, u)))) // (Inst t (
           class L)) -> (getFieldValue L l = Some u) -> (Red
           (get t l) (append t u))
     }
     case None => None
             }
      case Some(Pair(u,h)) => // Red_CGet
        Some(Pair(AGet(u,flab), RedCGet(t1,u,flab,h))); // (
            Red t u) -> (Red (get t l) (get t u))
      case None => None
    }
  case AOut(t1,label) => // Red_COut or Red_out ?
    red(t1) match {
      case Some(Pair(u,h)) => // Red_COut
        Some(Pair(AOut(u,label), RedCOut(t1,u,label,h)));
      case None =>
        // Looking for an expanded term that has the form (u
            label)
        lookupExp(t1,{
     case ANew(aterm,labelnew) => label == labelnew;
     case _ => false;
        }) match {
```

```
    case Some(Pair(ANew(u,clab),hexp)) => Some(Pair(u,
        RedOut(t1, u, clab, hexp)))
    case None => None
        }
   }
case _ => None
    }}

  // instance of
  def inst(t: ATerm, o: OLabel): Option[Inst] = {
    o match {
case Root() =>
  lookupExp(t,{
     case AThis(Root()) => true
     case _ => false
   }) match {
     case None => None
     case Some(Pair(AThis(Root()), exph)) => Some(InstRoot(
         t, exph))
   }
case OClasz(clab) =>
  lookupExp(t, {
     case ANew(u, newlab) => clab == newlab
     case _ => false
   }) match {
     case None => None
     case Some(Pair(ANew(u,newlab), exph)) => Some(
         InstClass(t, u, newlab, exph))
   }
    }
  }

  def wfTerm(t: ATerm): WFTerm = {
    t match {
case AThis(o) => WFThis(o)
case ANew(t1,label) =>
   val wft = wfTerm(t1); // (WF_Term t) ->
   val mkc = prog.getClasz.apply(label); // (getClass L) =
       (mkClass o s) ->
   val Some(tinsth) = inst(t1, mkc.owner); // (Inst t o) ->
  WFNew(t1, label, mkc.owner, mkc.t, wft, EquClass(label,
      mkc), tinsth)
case AOut(t1,label) =>
   val wft = wfTerm(t1); // (WF_Term t) ->
   val Some(tinsth) = inst(t1, prog.olabels(label.name));
       // (Inst t (class L)) ->
  WFOut(t1, label, wft, tinsth)
case AGet(t1,flabel) =>
   val wft = wfTerm(t1); // (WF_Term t) ->
```

```
    val mkf = prog.getField(flabel); // (getField l) = (
        mkField L u) ->
    val Some(tinsth) = inst(t1, OClasz(mkf.L)); // (Inst t (
        class L)) ->
WFGet(t1, flabel, mkf.L, mkf.t, wft, EquField(mkf.L,
    flabel, mkf.t), tinsth)
    }
  }

  def wfValuation(clab: CLabel, flab: FLabel, t: Term):
      WFVal = {
    // (getField l) = (mkField M u)
    val mkf = prog.getField(flab);
    // (WF_Term (append (This (class L))) t)
    val thisL = AThis(OClasz(clab));
    val wft = wfTerm(append(thisL, t));
    // (Inst (This (class L)) (class M))
    val olabM = OClasz(mkf.L);
    val Some(instthisLh) = inst(thisL, olabM);
    // (Red (append (This (class L)) u) u1) -> (Exp (append
        (This (class L)) t) u1)
    val Some(Pair(lred, lredh)) = lookupRed(append(thisL,
        mkf.t), x: ATerm => red(x) == None); // looking for
        the most reduced term
    val Some(Pair(u1, u1h)) = lookupExp(append(thisL, t), x
        : ATerm => lred == x); // looking for an exansion of
        (append (This (class L)) t) that equals u1
    WFVal(clab, flab, t, mkf.t, mkf.L, u1, EquField(mkf.L,
        flab, mkf.t), wft, instthisLh, lredh, u1h)
  }

  def wfProgram: WFProgram = {
    // for all getClass
    val classl = prog.getClasz.toList flatMap ({ case Pair(
        clab, mkClass(owner,t)) => t match {
case None => Nil
case Some(x) => List(wfTerm(append(AThis(owner),x)))
    }});
    // for all getField
    val fieldl = prog.getField.toList map ({ case Pair(flab
        ,mkf) => wfTerm(append(AThis(OClasz(mkf.L)), mkf.t))
        });
    // for all getFieldValue
    val fieldvaluel = prog.getFieldValue.toList flatMap ({
        case Pair(Pair(clab,flab), t) => t match {
case None => Nil
case Some(x) => List(wfValuation(clab,flab,x))
    }});
```

```
      WFProg(classl, fieldl, fieldvaluel, wfTerm(append(AThis
          (Root()), prog.getMain)))
    }

    wfProgram;
  }
}
```

Listing C.3: WFProofer.scala

Appendix D

An example of Well-Formedness Proof

We provide herein an example of well-formedness proof for the very simple program below. Even if it is one of the most basic SCALETTA program that one can write, the proof that it is well-formed is quite long. We also provide the formalization of the program in COQ.

```
class A {
  def t: !A;
}

class B extends !A {
  val t = !B;
}

def main: !A = !B;
```

```
(** Coq translation of the program **)
Require Calculus.

Module MyProgram.

  (** Class Label **)
  Inductive MyCLabel : Set :=
   | id_1_A   : MyCLabel
   | id_2_B   : MyCLabel
  .

  (** Field Label **)
  Inductive MyFLabel : Set :=
   | id_4_t   : MyFLabel
```

```
.

Definition CLabel: Set := MyCLabel.
Definition FLabel: Set := MyFLabel.

Definition CLabelDec: forall (L M: CLabel), {L = M} + {L <>
    M}.
Proof. decide equality. Qed.
Definition FLabelDec: forall (l m: FLabel), {l = m} + {l <>
    m}.
Proof. decide equality. Qed.

(** Class owner labels - O P Q **)
Inductive OLabel  : Set :=
| root            : OLabel
| class           : CLabel -> OLabel.

(** Terms - p q t u v w x y z **)
Inductive Term     : Set :=
  | this           : Term
  | new            : Term -> CLabel -> Term
  | get            : Term -> FLabel -> Term
  | out            : Term -> CLabel -> Term.

(** Field definitions **)
Inductive Field    : Set :=
  | mkField        : CLabel (** Field owner **)
                  -> Term   (** Field bound **)
                  -> Field.

(** Class definitions **)
 Inductive Class    : Set :=
  | mkClass        : OLabel       (** Class owner **)
                  -> option Term (** Class super **)
                  -> Class.

Definition getClass(L: CLabel): Class :=
  match L with
   | id_2_B    => (mkClass root (Some (new this id_1_A)))
   | id_1_A    => (mkClass root None)
  end.

Definition getField(l: FLabel): Field :=
  match l with
     | id_4_t     => (mkField id_1_A (new (out this id_1_A)
         id_1_A))
  end.
```

```
  Definition getFieldValue(L: CLabel)(m: FLabel): option Term
     :=
    match L,m with
       | id_2_B , id_4_t     => (Some (new (out this id_2_B)
          id_2_B))
       | _ , _ => None
    end.

  Definition getMain: Term :=
    (new this id_2_B)
    .
End MyProgram.


(** Proof lambda term **)

Require Typing.
Module MyTyping := Typing.SetProgram(MyProgram).
Import MyProgram.
Import MyTyping.
Lemma value: MyTyping.WF_Program.

(*** LEMMAS ***)
(*** proveGetFieldValue ***)
Inductive GetFieldValue: CLabel -> FLabel -> Term -> Prop :=
 | GFV0: (GetFieldValue id_2_B id_4_t (new (out this id_2_B)
    id_2_B))
.

Hint Resolve GFV0: scaletta.

Lemma impliesGetFieldValue: forall (L: CLabel) (l: FLabel) (t
   : Term),
 (getFieldValue L l) = (Some t) -> (GetFieldValue L l t).

Proof.
  induction L ; induction l ; simpl ; intros t H;
    (discriminate H) ||
    (injection H; intro H0; rewrite <- H0; auto with scaletta
       ).
Qed.

Lemma proveGetFieldValue:
  forall (P: CLabel -> FLabel -> Term -> Prop),
    (P id_2_B id_4_t (new (out this id_2_B) id_2_B)) ->

    (forall (L: CLabel) (l: FLabel) (t: Term),
     (getFieldValue L l) = (Some t) ->
     (P L l t)).
```

```
Proof.
  intros.
  apply GetFieldValue_ind; trivial.
  apply impliesGetFieldValue; trivial.
Qed.

(*** proveGetField ***)
Inductive GetField: CLabel -> Term -> Prop :=
 | GF0: (GetField id_1_A (new (out this id_1_A) id_1_A))
.

Hint Resolve GF0: scaletta.

Lemma impliesGetField: forall (l: FLabel) (L: CLabel) (t:
    Term),
 (getField l) = (mkField L t) -> (GetField L t).

Proof.
  induction l ; simpl ; intros L t H;
    (discriminate H) ||
      (injection H; intros H0 H1;
        rewrite <- H0; rewrite <- H1;
          auto with scaletta).
Qed.

Lemma proveGetField:
  forall (P: CLabel -> Term -> Prop),
    (P id_1_A (new (out this id_1_A) id_1_A)) ->

    (forall (l: FLabel) (L: CLabel) (t: Term),
     (getField l) = (mkField L t) ->
     (P L t)).
Proof.
  intros.
  apply GetField_ind; trivial.
  apply impliesGetField with (l := l); trivial.
Qed.

(*** proveGetSuper ***)
Inductive GetSuper: OLabel -> Term -> Prop :=
 | GS0: (GetSuper root (new this id_1_A))
.

Hint Resolve GS0: scaletta.

Lemma impliesGetSuper: forall (L: CLabel) (o: OLabel) (t:
    Term),
 (getClass L) = (mkClass o (Some t)) -> (GetSuper o t).
```

```
Proof.
  induction L ; simpl ; intros o t H;
    (discriminate H) ||
      (injection H; intros H0 H1;
        rewrite <- H0; rewrite <- H1;
          auto with scaletta).
Qed.

Lemma proveGetSuper:
  forall (P: OLabel -> Term -> Prop),
    (P root (new this id_1_A)) ->

    (forall (L: CLabel) (o: OLabel) (t: Term),
     (getClass L) = (mkClass o (Some t)) ->
     (P o t)).
Proof.
  intros.
  apply GetSuper_ind; trivial.
  apply impliesGetSuper with (L := L); trivial.
Qed.
(*** END LEMMAS ***)

(** Proof of well-formedness **)
exact(WF_Prog (proveGetSuper (fun o t => WF_Term (append (
   This o) t)) (WF_New (This root) id_1_A root None (WF_This
   root) (refl_equal (mkClass root None)) (Inst_Root (This
   root) (Exp_Refl (This root)))) )

 (proveGetField (fun L t => WF_Term (append (This (class L))
    t)) (WF_New (Out (This (class id_1_A)) id_1_A) id_1_A
    root None (WF_Out (This (class id_1_A)) id_1_A (WF_This (
    class id_1_A)) (Inst_Class (This (class id_1_A)) (This
    root) id_1_A (Exp_This id_1_A root None (refl_equal (
    mkClass root None))))) (refl_equal (mkClass root None)) (
    Inst_Root (Out (This (class id_1_A)) id_1_A) (Exp_Red (
    Out (This (class id_1_A)) id_1_A) (This root) (Red_Out (
    This (class id_1_A)) (This root) id_1_A (Exp_This id_1_A
    root None (refl_equal (mkClass root None))))))) )

 (proveGetFieldValue (fun L l t => WF_Valuation L l t) (
    WF_Val id_2_B id_4_t (new (out this id_2_B) id_2_B) (new
    (out this id_1_A) id_1_A) id_1_A (New (This root) id_1_A)
     (refl_equal (mkField id_1_A (new (out this id_1_A)
    id_1_A))) (WF_New (Out (This (class id_2_B)) id_2_B)
    id_2_B root (Some (new this id_1_A)) (WF_Out (This (class
     id_2_B)) id_2_B (WF_This (class id_2_B)) (Inst_Class (
    This (class id_2_B)) (This root) id_2_B (Exp_This id_2_B
    root (Some (new this id_1_A)) (refl_equal (mkClass root (
    Some (new this id_1_A))))))) (refl_equal (mkClass root (
```

```
    Some (new this id_1_A)))) (Inst_Root (Out (This (class
    id_2_B)) id_2_B) (Exp_Red (Out (This (class id_2_B))
    id_2_B) (This root) (Red_Out (This (class id_2_B)) (This
    root) id_2_B (Exp_This id_2_B root (Some (new this id_1_A
    )) (refl_equal (mkClass root (Some (new this id_1_A))))))
    ))) (Inst_Class (This (class id_2_B)) (This root) id_1_A
    (Exp_Trans (This (class id_2_B)) (New (This root) id_2_B)
     (New (This root) id_1_A) (Exp_This id_2_B root (Some (
    new this id_1_A)) (refl_equal (mkClass root (Some (new
    this id_1_A))))) (Exp_Ext (This root) (new this id_1_A)
    id_2_B root (refl_equal (mkClass root (Some (new this
    id_1_A))))))) (Red_CNew (Out (This (class id_2_B)) id_1_A
    ) (This root) id_1_A (Red_Out (This (class id_2_B)) (This
     root) id_1_A (Exp_Trans (This (class id_2_B)) (New (This
     root) id_2_B) (New (This root) id_1_A) (Exp_This id_2_B
    root (Some (new this id_1_A)) (refl_equal (mkClass root (
    Some (new this id_1_A))))) (Exp_Ext (This root) (new this
     id_1_A) id_2_B root (refl_equal (mkClass root (Some (new
     this id_1_A)))))))) (Exp_Trans (New (Out (This (class
    id_2_B)) id_2_B) id_2_B) (New (This root) id_2_B) (New (
    This root) id_1_A) (Exp_Red (New (Out (This (class id_2_B
    )) id_2_B) id_2_B) (New (This root) id_2_B) (Red_CNew (
    Out (This (class id_2_B)) id_2_B) (This root) id_2_B (
    Red_Out (This (class id_2_B)) (This root) id_2_B (
    Exp_This id_2_B root (Some (new this id_1_A)) (refl_equal
     (mkClass root (Some (new this id_1_A)))))))) (Exp_Ext (
    This root) (new this id_1_A) id_2_B root (refl_equal (
    mkClass root (Some (new this id_1_A))))))) )

 (WF_New (This root) id_2_B root (Some (new this id_1_A)) (
    WF_This root) (refl_equal (mkClass root (Some (new this
    id_1_A)))) (Inst_Root (This root) (Exp_Refl (This root)))
    )).
Qed.
```

Listing D.1: Proof of Well-Formedness

Bibliography

[AC05] Philippe Altherr and Vincent Cremet. Inner Classes and Virtual Types. *EPFL Technical Report IC/2005/013*, March 2005.

[coq] The coq proof assistant web page. *http://coq.inria.fr*.

[pic] An extensible compiler for the java programming language. *http://zenger.org/jaco/*.

[scaa] Scala web page. *http://scala.epfl.ch*.

[scab] Scaletta web page. *http://lamp.epfl.ch/∼paltherr/scaletta/*.