Inner Classes and Virtual Types

Philippe Altherr and Vincent Cremet

Ecole Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

EPFL Technical Report IC/2005/013, March 2005

Abstract. This paper studies the interplay between inner classes and virtual types. The combination of these two concepts can be observed in object-oriented languages like BETA or SCALA. This study is based on a calculus of classes and objects composed of a very limited number of constructs. For example the calculus has neither methods nor class constructors. Instead it has a more general concept of abstract inheritance which lets a class extend an arbitrary object. Thanks to an interpretation of terms as types the calculus also unifies type fields and term fields. The main contribution of this work is to show that typing virtual types in the presence of inner classes requires some kind of alias analysis and to formalize this mechanism with a simple calculus.

1 Introduction

The combination of inner classes and virtual types can be observed in several object-oriented programming languages like BETA [1] and SCALA [2]. The goal of this paper is to give a formal basis to study the interplay between inner classes and virtual types in these languages.

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 occurrence in a class needs to be reinterpreted in the context of a subclass. Because the context of the subclass usually gives more information than the context of the super class, it is possible to have a better knowledge about a type. In the presence of inner classes, the context of a class depends also on its statically enclosing class. The role of a type system is to make the most of this extra information.

Both concepts, inner classes and virtual types, have been studied and well-understood separately [3, 4] but little has been done on the formalization of their interaction. Our work focuses on this problem.

The paper is organized as follows. Section 2 is a reminder about inner classes. Section 3 introduces virtual types and describes informally the mechanism needed to type virtual types in the presence of inner classes. It reveals that the core of this mechanism is a kind of alias analysis. In Section 4 we introduce SCALETTA [1], a calculus of classes and objects whose goal is to type virtual types in the presence of inner classes. Because our primary goal does not require to consider mutable fields, we limit ourselves to a functional fragment where objects have no state and no identity like in [5]. The calculus has neither methods nor class constructors. Instead it has a more general concept of abstract inheritance which lets a class extend an arbitrary object.

[1] SCALETTA formalizes some important concepts of SCALA, hence its name.

This choice reduces greatly the number of evaluation rules. In Section 5, we give an interpretation of terms as types which lets us simulate type fields with normal term fields and introduce a type system based on the concept of abstract evaluation. We illustrate by an example how this type system performs the required alias analysis identified in Section 3. In Section 6, we extend the type system to address the problem of typing abstract inheritance. In Section 7 we discuss the undecidability of our type system and introduce the concept of typing strategies. In Section 8 we discuss the requirement of globally unique names in our calculus. We show that this requirement is equivalent to conceptually separate name resolution from typing. In section 9 we review related works. Finally we conclude with a summary of our contributions and present our plan for the future.

All examples presented in this paper have been typed-checked and evaluated using our SCALETTA compiler. Both the complete versions of the examples and the compiler can be found on the SCALETTA home page [6].

2 Inner Classes

Our calculus is about inner classes and virtual types. In this section, we introduce our terminology about inner classes and explain what we call an inner class. We remind also some, maybe not so well-known, facts about inner classes.

2.1 Terminology

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. These rights pose some interesting and non-trivial issues. However these issues are beyond the scope of this paper. Here, we are only interested in the additional issues posed by the presence of a current enclosing instance in inner classes. The rest of this section illustrates these issues with some examples written in JAVA [7].

2.2 Enclosing Instances

In JAVA, any non-static class declared within some class `C` is an inner class. Within the inner class, the current enclosing instance is denoted by the expression `C.this` which is of type `C`. The code below declares an inner class `I` nested in a class `R`. It makes explicit the presence of a current enclosing instance of type `R` by declaring a field `outerI` of that type and initializing it with that instance.

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

This example explicitly declares a field `outerI` that holds the current enclosing instance. However, every inner class really has a hidden field that holds this instance and the syntax `C.this` is just a way to access this hidden field. In fact, an inner class is nothing else than a static nested class with an additional field holding the current enclosing instance. We call this additional field *the outer field* of the inner class.

When an inner class is declared within another inner class, it can access the current instances of both of its enclosing classes. For example, within class `M` in the code below, the expression `R.this` denotes the current instance of the (indirectly) enclosing class `R`. This instance is, by definition, equal to `R.this` evaluated within class `I`. In other words, in class `M`, `R.this` is equal to `this.outerM.outerI`.

```
public class R {
  class I {
    final R outerI = R.this;
    class M {
      final I outerM = I.this;
    }
  }
}
```

So, although an inner class may have access to the current instance of several enclosing classes, a single outer field per class is sufficient to access all these instances. In the general case, within a class $\mathtt{C}_0$ nested in a class $\mathtt{C}_1$, ..., nested in a class $\mathtt{C}_n$, the expression $\mathtt{C}_i$`.this` is equal to `this.outer`$\mathtt{C}_0$ ... `.outer`$\mathtt{C}_{i-1}$. Therefore, the syntax `C.this` would be superfluous if the outer fields were not hidden and were automatically initialized. It could always be replaced by a succession of outer field selections.

2.3 Instance Creations

To create a new instance of an inner class `D`, an instance of its enclosing class `C` has to be provided. In Java, the syntax *`expr`*`.new D(`*`args`*`)` is used to that effect. It creates a new instance of class `D` whose enclosing instance is the result of the expression *`expr`*. The enclosing instance may be omitted if the instance creation is enclosed, possibly indirectly, in a subclass `E` of `C`. In that case, the expression `new D(`*`args`*`)` is equivalent to the expression `E.this.new D(`*`args`*`)`. We illustrate this by augmenting the class `R` above with the two fields declared below. The value `i` is an instance of class `I` whose enclosing instance is the current instance of class `R` (the expression `new I()` really means `this.new I()`) and the value `m` is an instance of class `M` whose enclosing instance is the value `i`.

```
  final I   i = new I();
  final I.M m = i.new M();
```

Every inner class introduces a single outer field, but an instance of an inner class may have several outer fields because it inherits one from each inner class it is an instance of. For example, every instance of the class `J` declared below has two outer fields, namely `outerI` and `outerJ`, and every instance of the class `N` has the two outer fields `outerM` and `outerN`.

```
public class R {
  class I {
    final R outerI = R.this;
    class M {
      final I outerM = I.this;
    }
    class N extends M {
      final I outerN = I.this;
      N(I i) { i.super(); }
    }
  }
  class J extends I {
    final R outerJ = R.this;
  }
  final I   i = new I();
  final J   j = new J();
  final I.N n = i.new N(j);
}
```

For the same reason instance creations of inner classes require an enclosing instance, super constructor calls of inner classes also require an enclosing instance. These super constructor calls have the syntax `expr.super(args)` where *expr* must evaluate to an instance of the enclosing class `C` of the super class. As with instance creations, the enclosing instance may be omitted if the super constructor call is enclosed, possibly indirectly, in a subclass `E` of `C`. In that case, the expression `super(args)` is equivalent to `E.this.super(args)`.

2.4 Aliasing

In the code above, the class `J` has no explicit constructor; it gets a default constructor containing a call to the super constructor, `super()`, which is here equivalent to `R.this.super()`. This implies that for any instance of class `J`, its two outer fields `outerI` and `outerJ` hold exactly the same value. The constructor of class `N` specifies that the enclosing instance of its superclass `M` is its argument `i`. Therefore, the two outer fields of a given instance of class `N` may hold different values. For example, for the value `n`, `n.outerN` is equal to `i` while `n.outerM` is equal to `j`. In fact, the two values are even of different types.

The next example illustrates that it is sometimes possible to statically establish that two outer fields hold the same value. It implies that the two fields can be assumed by the compiler to share their typing knowledge.

```
public class R {
  class A {
    final R outerA = R.this;
    class X           { final A outerX = A.this; }
  }
  class B extends A {
    final R outerB = R.this;
```

```
    class Y extends X { final B outerY = B.this; }
  }
}
```

Instances of the class `Y` declared above have two outer fields: `outerX` and `outerY`. One can establish that both will always hold the same value, but both are not perfectly equivalent; `outerY` is of type `B` while `outerX` is only of type `A`. However, we know that `outerX` holds an instance of class `B`. Thus, we know that each time something is accessed via `outerX`, it is accessed on an instance of class `B`. The next section will show that this information is crucial in the context of virtual types.

2.5 Summary

To sum up, an inner class can be viewed as a top-level class with an additional field (the outer field) holding the current enclosing instance. This instance must be provided to all instance creations and all super constructor calls. An instance of an inner class inherits one outer field from each inner class it is an instance of. All those fields may hold different values, but sometimes it is possible to formally establish that some of them necessarily hold the same value.

3 Virtual Types

In this section we review virtual types and identify the kind of analysis a type system has to perform when virtual types are used in combination with inner classes.

3.1 Definition

In some object-oriented languages, it is possible to declare abstract type members, i.e. type members that are bounded by a type but have no exact type value. These members may then be given different type values in different subclasses. This means that the exact value of such a type member depends on the exact class of the value from which it is selected. These type members are called *virtual types*. We illustrate virtual types with the following example written in Scala:

```
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 `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 a subtype of `Object`, it would be illegal to assign `"foo"` to `x` in class `M` because in subclasses of M, T may be assigned any subtype of Object and the value of x has to be an instance of that type. In the subclass N, T is assigned the type String. As in SCALA type assignments may not be overridden in subclasses, it is possible to assign "foo" to x in class N.

3.2 Virtual Types with Inner Classes

We show now that typing virtual types in the presence of inner classes requires some kind of alias analysis. To illustrate this we consider a SCALA version of the example of Section 2.4 augmented with a type member T in class A and a field x in class X.

```
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";
  }
}
```

The code above and in particular the assignment of field x in class Y is well-typed. This might be a bit surprising because the field x is declared in class X with the type T and T is an abstract type member of class A. The exact value of T is not known, at least not in class X. So, how could one assign "foo" to x in one of its subclasses? The answer comes from the observation we made in Section 2.4: for all instances of class Y, the fields outerX and outerY hold the same value. The field x is declared with the type T. In class X, T is a shorthand for A.this.T and by definition A.this is equal to this.outerX, so T is equal to this.outerX.T. As for any instance of Y, outerX is equal to outerY, we conclude that in class Y, the field x has the type this.outerY.T which is obviously equal to String. The assignment of "foo" to x in class Y is therefore well-typed.

This example illustrates the fact that typing virtual types in the presence of inner classes requires some kind of alias analysis on outer fields. In this example, it is possible to establish that x has 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 T is bound by Object.

3.3 Outer This vs. Outer Fields

In the SCALA compiler, the alias analysis mentioned above is performed by a rather complex method called asSeenFrom. This method takes as arguments a type T, the class C where this type occurs and a subclass D of C. It returns the reinterpretation of T in class D. Here is an application of that method:

```
asSeenFrom(A.this.T, X, Y)  →  String
```

To obtain this result the `asSeenFrom` method first does some alias analysis to establish that the expression `A.this` occurring in class `X` once reinterpreted in class `Y` denotes the same value as the expression `B.this` interpreted in class `Y`. From this `asSeenFrom` determines that `A.this.T` reinterpreted in class `Y` is equal to `B.this.T`. Finally, it applies the fact that `T` is given the type `String` in class `B` and returns `String`.

One reason the method `asSeenFrom` is rather complex is that its input types are usually well-formed only in the class where they occurred. For example, the type `A.this.T` is not well-formed in class `Y` because `A` is not an enclosing class of `Y`. It is also possible to construct examples where the input type is well-formed in both of the input classes but has a different meaning. This has to do with the fact that in general expressions of the form `C.this` may not be moved from the body of a class to the body of one of its subclasses.

In our calculus, we avoid that problem by translating all expressions of the form `C.this` into a sequence of outer fields selections on `this`. This has the great advantage that the resulting expressions are well-formed in the class where they occur and in all of their subclasses and have the same meaning in all those classes. For example, the type `this.outerX.T` of the field `x` in class `X` is also well-formed in class `Y` and has the same meaning than in class `X`.

3.4 Parameterized Classes

In this section, we show that the combination of parameterized classes and inner classes induces the same kind of analysis as the combination of virtual types and inner classes.

Parameterized classes have been presented as an alternative to virtual types [8]. For instance, the code example of the previous subsection can also be written in a language with parameterized classes and inner classes like JAVA 5.0. The idea of the translation is to let an abstract type member of a class be a type parameter of this class.

```
class A<T extends Object> {
  abstract class X {
    final A<T> outerX = A.this;
    abstract T x();
  }
}
class B extends A<String> {
  class Y extends X {
    final B outerY = B.this;
    String x() { return "foo"; }
  }
}
```

The code above is well-typed JAVA 5.0 code. In particular, the method `x` in class `Y` effectively implements the method `x` declared in class `X`. This is true for exactly the same reasons the assignment of the field `x` was well-typed in the SCALA example. The well-formedness proof requires exactly the same alias analysis to determine that the fields `outerX` and `outerY` hold the same value for any instance of class `Y` and that therefore the type `T` occurring in class `X` reinterpreted in class `Y` is equal to the type `String`.

This observation shows that our calculus which can be used to type virtual types in the presence of inner classes can also be used to understand the interaction between inner classes and parameterized classes or at least all aspects related to alias analysis.

3.5 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 in 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 translation 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`.

4 Untyped Calculus

This section describes the untyped fragment of our calculus and explains its semantics. The syntax and all deduction rules are summed up in Figure 1.

4.1 Syntax

A Scaletta program consists of a list of class declarations and a main expression. Given a meta-variable x, we use the notation $\overline{x}$ to represent a potentially empty sequence $x_1, \ldots, x_n$, of elements denoted by x. Each class has a name, zero or one parent and a list of field valuations. 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.

Our calculus implements inheritance through delegation. This means that any instance c of a 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 new instances 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. The instance creation $t\,!\,C$ corresponds to the Java expression t`.new` C`()`. It creates a new instance of class C with the enclosing instance t. In other words, it creates a new instance of class C whose implicit outer field is initialized with t. The outer field selection $t\,@\,C$ corresponds to the Java expression t`.outer`C where `outer`C denotes the implicit outer

Syntax

Class name	B, C		
Field name	f		
Class declaration	D	$::= \texttt{class}\ C\ \texttt{extends}\ p\ \{\,\overline{d}\,\}$	
Class parent	p	$::= t \mid \texttt{nothing}$	
Class member	d	$::= \texttt{field}\ f = t$	field valuation
Term	t, u	$::= \texttt{this}$	current instance
		$\mid\ t\ \texttt{!}C$	instance creation
		$\mid\ t\ .\ f$	field selection
		$\mid\ t\ \texttt{@}C$	outer field selection
Program	P	$::= \overline{D}\ t$	
Evaluation context	E	$::= \langle\rangle \mid E\ \texttt{!}C \mid E\ .\ f \mid E\ \texttt{@}C$	
Values	v	$::= \texttt{this} \mid v\ \texttt{!}C$	

Auxiliary Relations

(Extends) $$\frac{\texttt{class}\ C\ \texttt{extends}\ p\ \{\,\overline{d}\,\}}{C \text{ extends } p}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ p\ \{\,\overline{d}\,\}}{C \text{ declares } d_i}$$ (Declares)

Reduction

(→_Field) $$\frac{t \ <\ C \qquad C \text{ declares } (\texttt{field}\ f = u)}{t\ .\ f \rightarrow [t/\texttt{this}]u}$$

(→_Outer) $$\frac{t \prec u\ \texttt{!}C}{t\ \texttt{@}C \rightarrow u}$$

(→_Context) $$\frac{t \rightarrow u}{E\langle t\rangle \rightarrow E\langle u\rangle}$$

Expansion

$$\frac{t \ <\ C \qquad C \text{ extends } u}{t \prec [t\ \texttt{@}C/\texttt{this}]u}$$ (≺_Ext)

$$\frac{t \rightarrow u}{t \prec u}$$ (≺_Red)

+ reflexivity (≺_Refl)
+ transitivity (≺_Trans)

Instance

$$\frac{t \prec u\ \texttt{!}C}{t \ <\ C}$$ (<_New)

Fig. 1. The untyped calculus

field of class C. It returns the contents of that field, provided t is an instance of class C. We remind here that a value has as many outer fields as inner classes it is an instance of. That is the reason we need the name C; it identifies the class whose outer field should be returned. In some sense, the operation $t\,\texttt{@}C$ is the opposite of $t\,\texttt{!}C$; it extracts from an instance of class C the enclosing instance that was used to create it. One could expect that for any term t and any class C, the expression $t\,\texttt{!}C\,\texttt{@}C$ is equal to t. Later in this section we will see that this is indeed true.

4.2 Semantics

The semantics of SCALETTA is defined by three inductive relations and two auxiliary relations. All are implicitly parameterized by the list of class declarations of the program; when a class declaration D appears in the premises of a rule, it simply means that D belongs to this list.

Among the three semantics relations, there is a reduction relation $t \to u$ which states that t reduces to u. The two others are the expansion relation $t \prec u$ which states that u is a delegate of t and the instance relation $t < C$ which states that t is a (direct or indirect) instance of class C.

Values are terms that consist only of instance creations. The initial `this` of values and also of all terms that appear in the deduction rules of the three semantics relations designates the unique instance of the implicit root class.

The expansion relation extends the notion of delegate of a term t to include not only the direct delegate of new instances ($\prec$_Ext), but also the term t itself ($\prec$_Refl), all delegates of any of its delegate ($\prec$_Trans) and the reduction of any of its delegate ($\prec$_Red). The outer field selection $t\,\texttt{@}C$ in rule ($\prec$_Ext) expresses the fact that the parent term t of a class C has to be evaluated in the context of the enclosing class of C (i.e. in the parent term, `this` designates the current enclosing instance of class C). We use the notation $[t/\texttt{this}]u$ to represent the term obtained by substituting t for `this` in u.

The instance relation consists of the single rule ($<$_New) which states that a term t is an instance of a class C if one of its delegates is a (direct) new instance of class C. Strictly speaking, this relation is not really necessary here; its single rule could easily be inlined in the two other relations. We introduce it in anticipation of the typed calculus which adds two other deduction rules.

The reduction relation imposes no order on evaluation, as expressed by the deduction rule ($\to$_Context). In this rule the notation $E\langle t\rangle$ represents the term obtained from the context E by replacing the hole $\langle\rangle$ with the term t. Field selections are reduced the same way as parameterless methods are evaluated in standard object-oriented languages ($\to$_Field) by a lookup of the field starting from the receiver object. Note that, as our calculus is purely functional, this behavior is indistinguishable from one where fields are evaluated only once and cached. The rule ($\to$_Outer) expresses the fact that the outer fields of a value are split over its delegates; each delegate stores the contents of one outer field, namely the one of the class it is a direct instance of. It expresses also the fact that given the exact class of a value t and the contents of the outer field of that class, the contents of all other outer fields can be computed simply by recomputing the delegates of t.

4.3 Examples

First, we prove that, as stated earlier, for any term t and any class C, the term $t\,\texttt{!}C\,\texttt{@}C$ reduces to t. Here is the proof:

$$(\rightarrow\_\text{Outer})\dfrac{(\prec\_\text{Refl})\dfrac{}{t\,\texttt{!}C \prec t\,\texttt{!}C}}{t\,\texttt{!}C\,\texttt{@}C \rightarrow t}$$

Now, let us try to encode positive integers with a base class `Int` and two subclasses `Zero` and `Succ`. The idea is to encode 0 with the value `this!Zero` and any strictly positive integer n with an instance of class `Succ` whose enclosing instance stores the predecessor of n. Thus, any positive integer n is represented by the value `this!Zero` followed by n instance creations of class `Succ`. For example, the number 2 is represented by the value `this!Zero!Succ!Succ`. For now, we equip our integers with only two fields: `pred` and `succ` which return respectively the predecessor and the successor of the receiver integer.

```
class Int extends nothing {
  field succ = this!Succ;
  class Succ extends this@Int!Int {
    field pred = this@Succ;
  }
}
class Zero extends this!Int {
  field pred = this;
}
```

Note that although in the calculus all classes are declared at the same level, in our code examples we nest classes that are logically nested. For example, in the code above, the class `Succ` is nested in class `Int` because we expect that the enclosing instance of any instance of class `Succ` will always be an instance of class `Int`.

Now, let us try to reduce the term `this!Zero.succ.pred`. We start by proving that `this!Zero` is an instance of class `Int`:

$$(\rightarrow\_\text{Outer})\dfrac{(\prec\_\text{Ext})\dfrac{(<\_\text{New})\dfrac{(\prec\_\text{Refl})\dfrac{}{\texttt{this!Zero} \prec \texttt{this!Zero}}}{\texttt{this!Zero} < \texttt{Zero}} \quad \texttt{Zero}\ \text{extends}\ \texttt{this!Int}}{\texttt{this!Zero} \prec \texttt{this!Zero@Zero!Int}}}{\texttt{this!Zero} < \texttt{Int}}$$

As class `Int` declares that field `succ` is equal to `this!Succ`, we may apply the rule ($\rightarrow$_Field) to reduce `this!Zero.succ` to `this!Zero!Succ`. This is an instance of class `Succ` which declares that field `pred` is equal to `this@Succ`. We may therefore again apply the rule ($\rightarrow$_Field) to reduce `this!Zero!Succ.pred` to `this!Zero!Succ@Succ` which, by applying our previous result about terms of the form $t\,\texttt{!}C\,\texttt{@}C$, reduces to `this!Zero`. Altogether, with some additional applications of the rule ($\rightarrow$_Context), this lets us conclude that `this!Zero.succ.pred` reduces to `this!Zero`.

4.4 Syntactic Sugar

In order to make our next examples a bit easier to read, we introduce some syntactic sugar. First of all, we will omit the `extends` clause of classes that extend nothing. We also introduce the JAVA syntax C`.this` to designate the current instance of an enclosing class C. The translation of such an expression into a sequence of outer field selections is explained in Section 2.2. To terminate, we make the initial `this` of a term optional.

Here is the integer example rewritten with the newly introduced syntactic sugar.

```
class Int {                            // "extends nothing" -> ""
  field succ = !Succ;                  // "this"      -> ""
  class Succ extends Root.this!Int { // "this@Int"  -> "Root.this"
    field pred = Int.this;             // "this@Succ" -> "Int.this"
  }
}
class Zero extends !Int {              // "this"      -> ""
  field pred = this;
}
```

4.5 Methods

Methods are not part of the calculus but they can be encoded using classes. An implementation of a method m is encoded by an auxiliary class M and a field m that returns a new instance of the class M. The class M has one abstract field for each parameter of m and a result field whose value is the encoding of the body of the method. This encoding simply replaces all references to parameters of m by references to the corresponding fields of class M. The values of the parameter fields are provided when the method is applied.

To illustrate this, we define for our integers a method `add` that computes the sum of the receiver object and its argument. We use a field `that` to encode the method parameter and a field `result` for its result.

The implementation of `add` in class `Zero` is simple; the method simply returns its parameter. We augment the class `Zero` with the following declarations.

```
class AddInZero {
  field result = that;
}
field add = !AddInZero;
```

Note that in the untyped calculus abstract fields are not declared at all. That explains the occurrence of the undeclared field `that` in class `AddInZero`.

The implementation of `add` in class `Succ` is a bit more difficult because it involves a method application, namely a recursive call to itself. For now, let us assume that we know how to encode method applications. This leads to the following declarations in class `Succ` where `add(that.succ)` is syntactic sugar for the application of method `add` to the argument `that.succ`.

```
class AddInSucc {
  field result = Succ.this.pred.add(that.succ);
}
field add = !AddInSucc;
```

An application $t\,\texttt{.}\,m(\overline{u})$ of a method m with the receiver object t and arguments $\overline{u}$ is encoded by the term $\texttt{this}\,\texttt{!}N\,\texttt{.}\,r$ where N is an auxiliary class and r the result field name of the method m. The class N extends the term $t\,\texttt{.}\,m$ and contains a field valuation for each argument u_i.

We can now rewrite the class `AddInSucc` without syntactic sugar for method application:

```
class AddInSucc {
  class Apply extends Succ.this.pred.add {
    field that = AddInSucc.this.that.succ;
  }
  field result = !Apply.result;
}
```

All instances of class `Succ` have the same single delegate `this!Int`. That is not true for the class `Apply` for which the value of the parent varies from one instance to the other. For example it can be shown that the only delegate of `1!AddInSucc!Apply` is `0!AddInZero` while the only delegate of `2!AddInSucc!Apply` is `1!AddInSucc`. Such a behavior is only possible because in our calculus classes extend arbitrary terms instead of classes. We call this mechanism *abstract inheritance*. Abstract inheritance is the key element that lets us encode methods.

4.6 Blocks

As a second demonstration of the simplicity for encoding classical high-level programming constructs in SCALETTA, we show now how we can encode blocks. We introduce the syntactic sugar $\{\overline{D}\,\overline{d}\ \ t\}$ for expressing a block consisting of a list of definitions (classes and fields) $\overline{D}\,\overline{d}$ and of a main expression t representing the value of the block. Such a block expression is translated into the class

$$\texttt{class}\,C\,\{\overline{D}\quad\overline{d}\quad\texttt{field}\,result\,=\,t\}$$

and the expression

$$\texttt{this}\,\texttt{!}C\,\texttt{.}\,result$$

where C is a fresh class name and *result* a fresh field name for accessing the result of the block.

4.7 Functions

Using inner classes, methods and blocks, it is possible to encode functions as objects containing a method `apply` and function applications as calls to this method. It follows that our calculus has first-class functions and therefore can trivially encode the lambda-calculus.

We formally define here the translation $\langle M \rangle$ from a term M of the lambda-calculus into an untyped SCALETTA program $\overline{D}\, t$ consisting of a list of classes $\overline{D}$ and a main term t. For expressing the result of the translation, we use the already introduced syntactic sugar for methods and blocks, and we spatially nest classes that are logically nested.

$$\text{(Variable)} \quad \frac{}{\langle x \rangle = x}$$

$$\text{(Application)} \quad \frac{\langle M \rangle = \overline{D}\, t \qquad \langle N \rangle = \overline{D}'\, u}{\langle M\ N \rangle = \overline{D}\,\overline{D}'\ \ t.\mathrm{apply}(u)}$$

$$\text{(Abstraction)} \quad \frac{\langle M \rangle = \overline{D}\, t \qquad C \text{ fresh class name}}{\langle \lambda x.M \rangle = \ \text{class } C\ \{ \text{ method apply(x)} = \{\ \overline{D}\ \text{t}\ \}\ \}\ \ \text{this!}C}$$

For instance the encoding of the term $\lambda f.\lambda x.f\ x$ is

```
class C1 {
  method apply(f) = {
    class C0 {
      method apply(x) = {
        f.apply(x)
      }
    }
    this!C0
  }
}
this!C1
```

4.8 Safety and Confluence

The calculus presented here is neither safe nor confluent. It is unsafe because the reduction of field and outer field selections may both be stuck. The reduction of a field selection t . f is stuck if the value denoted by the term t has no valuation for the field f. The reduction of an outer field selection t @C is stuck if the value denoted by the term t is not an instance of class C. These two issues are addressed by Section 5 which augments the calculus with type annotations and presents a well-formedness predicate.

The calculus is not confluent for two reasons. Firstly, a value may inherit a field valuation for a given field f from several classes. If the field f is selected on such a value, the rule (→_Field) does not specify which valuation should be used. Therefore, anyone can be used. Secondly, it is possible to build values that inherit several times from some class C but with different enclosing instances. If the outer field of class C is selected on such a value, the rule (→_Outer) does not specify which enclosing instance should be used. So, here again, anyone can be used. Both problems could be solved by somehow ordering the inherited field valuations and enclosing instances and modifying the rules (→_Field) and (→_Outer) to pick the first one. However, we do not do that because our well-formedness predicate described in Section 5 can not tolerate programs where such things may arise. In Section 6 we describe the additional rules needed to reject such programs.

4.9 Alias Analysis

We terminate this section with an example showing that our calculus can indeed be used to do some alias analysis. The code below is the Scaletta version of the example given in Section 2.4.

```
class A {
  class X {}
}
class B extends !A {
  class Y extends !X {}    // "!X" means "this!X"
}
```

As in Scaletta all classes have an implicit and accessible outer field, the explicit outer fields have been omitted. In Section 2.4, we established that for any instance of class `Y`, its two outer fields hold the same value. We show here that with the given rules it is already possible to formally establish that, for a given instance of class `Y`, its two outer field hold the same value.

We do here the proof for the value `this!B!Y`. We have to show that `this!B!Y@X` and `this!B!Y@Y` denote the same value. The formal proof below demonstrates that the first term reduces to the second one.

$$
(\rightarrow\_\mathrm{Outer})\dfrac{(\prec\_\mathrm{Ext})\dfrac{\begin{array}{c}(<\_\mathrm{New})\dfrac{(\prec\_\mathrm{Refl})\dfrac{}{\mathtt{this\,!B\,!Y} \prec \mathtt{this\,!B\,!Y}}}{\mathtt{this\,!B\,!Y} < \mathtt{Y}} \\ \mathtt{Y}\ \text{extends}\ \mathtt{this\,!X}\end{array}}{\mathtt{this\,!B\,!Y} \prec \mathtt{this\,!B\,!Y\,@Y\,!X}}}{\mathtt{this\,!B\,!Y\,@X} \rightarrow \mathtt{this\,!B\,!Y\,@Y}}
$$

5 Type System

In this section we introduce a type system for the untyped calculus presented in the previous section and show how it can be used to perform the alias analysis required to type the example described in Section 3.2.

5.1 Types

Typing a program written in an object-oriented language requires to approximate abstract fields. In our calculus we follow the classical solution that consists in attaching a type to a field.

Abstract type fields and abstract term fields share the property of being *virtual*; their value depends on the exact class of the value from which they are selected. For term fields the computation of their value takes place at runtime and is called *late binding* or *polymorphism*. For type fields this computation happens at compile time.

To keep our calculus as small as possible we want to factorize the mechanism that governs the virtuality of term and type fields. To achieve this, we choose the most naive solution we can think of: using terms as types.

$$\boxed{\text{Type } T ::= t}$$

However we have to find an interpretation of terms as types. Here is this interpretation: a term t is of type u, where u is another term, if the value resulting from the evaluation of u is a delegate of the value resulting from the evaluation of t. It makes sense to approximate objects by their delegates because they inherit field valuations and enclosing instances from them.

The unification of types and terms lets us simulate type fields with term fields, as illustrated later in the example of Section 5.5.

5.2 Annotations

To make type-checking feasible we add to a program a list of top-level annotations as presented in Figure 2.

Class annotation	$A ::= \texttt{class}\ C\ \texttt{inside}\ B$
Field annotation	$a\ ::= \texttt{field}\ f : T\ \texttt{inside}\ B$
Program with annotations	$\overline{A}\ \overline{a}\ P$

Fig. 2. Annotations

There are two kinds of annotations, one for classes and another for fields. A class annotation `class` C `inside` B constraints the enclosing instance of an instance of C to be an instance of B. In our code examples this requirement is implicitly expressed by nesting classes. Classes declared at the top-level are implicitly nested in a class `Root` where `Root` is a distinguished class name with no corresponding class declaration.

A field annotation `field` $f : T$ `inside` B declares a bound T for the field f. The `inside` clause states that field f may only be selected on instances of class B. In our code examples this clause is implicitly expressed by nesting the field annotation in class B.

5.3 Abstract Evaluation

In Section 3 we have seen that typing virtual types in the presence of inner classes requires to establish some equalities between outer fields. In the example of Section 4.9 we have established the equality between two outer fields of *a particular* instance of `Y` by evaluating them in our calculus. To apply this result to typing we have to establish that property *for all* instances of `Y`. A priori we can not use the same technique because we can not evaluate the two outer fields of all instances of `Y` and check that they denote the same value. We have to abstract over the particular instance of `Y`, but we want to keep the appealing principle of establishing equality by evaluation.

In conclusion we need to evaluate terms in a partially unknown context, i.e. a context where the exact class of the current instance is not known, but also where

Abstract Reduction

$$(\rightarrow^{abs}\_\mathrm{Field}) \quad \frac{B \vdash t \ < \ C \qquad C \text{ declares } (\texttt{field}\ f = u)}{B \vdash t\,.\,f \rightarrow [t/\texttt{this}]u}$$

$$(\rightarrow^{abs}\_\mathrm{Outer}) \quad \frac{B \vdash t \prec u\,\texttt{!}C}{B \vdash t\,\texttt{@}C \rightarrow u}$$

$$(\rightarrow^{abs}\_\mathrm{Context}) \quad \frac{B \vdash t \rightarrow u}{B \vdash E\langle t\rangle \rightarrow E\langle u\rangle}$$

Abstract Expansion

$$\frac{B \vdash t \ < \ C \qquad C \text{ extends } u}{B \vdash t \prec [t\,\texttt{@}C/\texttt{this}]u} \quad (\prec^{abs}\_\mathrm{Ext})$$

$$\frac{B \vdash t \rightarrow u}{B \vdash t \prec u} \quad (\prec^{abs}\_\mathrm{Red})$$

$+$ reflexivity $(\prec^{abs}\_\mathrm{Refl})$

$+$ transitivity $(\prec^{abs}\_\mathrm{Trans})$

$$\frac{\texttt{field}\ f : u\ \texttt{inside}\ C}{B \vdash t\,.\,f \prec [t/\texttt{this}]u} \quad (\prec^{abs}\_\mathrm{Def})$$

Abstract Instance

$$\frac{B \vdash t \prec u\,\texttt{!}C}{B \vdash t \ < \ C} \quad (<^{abs}\_\mathrm{New})$$

$$\frac{}{B \vdash \texttt{this} \ < \ B} \quad (<^{abs}\_\mathrm{This})$$

$$\frac{\texttt{class}\ C\ \texttt{inside}\ C'}{B \vdash t\,\texttt{@}C \ < \ C'} \quad (<^{abs}\_\mathrm{Outer})$$

Fig. 3. Abstract Evaluation

some fields are abstract. Such an *abstract evaluation* is easily obtained by adding a class context B to each evaluation rule. The class context specifies the class in which the terms have to be interpreted. In other words the class B denotes the class `this` is the current instance of.

The relations that constitutes abstract evaluation are summarized in Figure 3. They are implicitly parameterized by the lists of class declarations, class annotations and field annotations that constitute the program.

Compared to evaluation there are three new rules: one expansion rule and two instance rules. The first new instance rule ($<^{abs}$_This) tells us that in the class context B the current instance `this` is indeed an instance of B.

The second new instance rule ($<^{abs}$_Outer) and the new expansion rule ($\prec^{abs}$_Def) have similar roles. The former lets us approximate the value of the outer field of a class C to an instance of the enclosing class of C. The latter lets us approximate the value of a field to its declared bound. This rule can compensate the absence of a valuation for a field.

We want now to prove two claims; first that abstract evaluation is a generalization of evaluation and second that abstract evaluation can be used to compute in a partially unknown context.

The first claim is equivalent to the theorem below, whose proof is trivial, and the second claim is demonstrated by Section 5.5 which computes the value of a field in a partially unknown context.

Theorem 1. $t \to u$ *implies* $\texttt{Root} \vdash t \to u$

5.4 Well-formedness

Abstract evaluation is the core of our type system. We present now typing rules that make use of abstract evaluation to express the well-formedness of terms, class declarations, field declarations and programs. The rules are summarized in Figure 4. They are implicitly parameterized by the lists of class declarations, class annotations and field annotations that constitute the program.

A particularity of this type system is that the relation that checks if a term is well-formed does not assign a type to this term. That is not necessary in our calculus because as types are terms, the more accurate type that can be given to a term is itself.

The well-formedness relation for terms is parameterized by a class B which has the same meaning as the one appearing in the abstract evaluation rules.

Note that in the rule ($\diamond$_Class) for classes the parent clause of the class must be interpreted in the context of the enclosing class.

In the rule ($\diamond$_Val) for field valuations, the premise $B \vdash u \to^* u'$ contains the central idea that the bound u of a field f is reinterpreted in the context of the class B containing the field valuation and that we get the more accurate value u'.

The type system presented above is not sound. For instance, it does not ensure that a term t has no abstract fields when used as an enclosing instance, like in $t\,\texttt{!}C$, or as prefix of a selection, like in $t\,\texttt{.}f$ or $t\,\texttt{@}C$. This problem is solved in Section 6 where we address the problem of typing abstract inheritance. However it should be clear that this type system can already reject a lot of invalid programs, namely those that select fields or outer fields on terms that do not even inherit a declaration for that fields. Our problem is that we accept programs that select fields and outer fields on terms that declare them but not implement them.

Term w.f.

$$(\diamond\_\mathrm{This})\quad \frac{}{B \vdash \texttt{this}\ \diamond}$$

$$(\diamond\_\mathrm{New})\quad \frac{\begin{array}{c} B \vdash t \diamond \\ B \vdash t\ <\ C' \\ \texttt{class}\ C\ \texttt{inside}\ C' \end{array}}{B \vdash t\,\texttt{!}C \diamond}$$

$$(\diamond\_\mathrm{Get})\quad \frac{\begin{array}{c} B \vdash t \diamond \\ B \vdash t\ <\ C' \\ \texttt{field}\ f : T\ \texttt{inside}\ C' \end{array}}{B \vdash t\,.\,f \diamond}$$

$$(\diamond\_\mathrm{Outer})\quad \frac{\begin{array}{c} B \vdash t \diamond \\ B \vdash t\ <\ C \end{array}}{B \vdash t\,\texttt{@}C \diamond}$$

Class w.f.

$$\frac{\begin{array}{c} \texttt{class}\ C\ \texttt{inside}\ B \\ p = t \Rightarrow B \vdash t \diamond \\ C \vdash \overline{d} \diamond \end{array}}{\vdash \texttt{class}\ C\ \texttt{extends}\ p\ \{\,\overline{d}\,\} \diamond}\quad (\diamond\_\mathrm{Class})$$

Field w.f.

$$\frac{\begin{array}{c} \texttt{field}\ f : u\ \texttt{inside}\ C' \\ C' \vdash u \diamond \\ B \vdash \texttt{this}\ <\ C' \\ B \vdash t \diamond \\ B \vdash u \rightarrow^{*} u' \\ B \vdash t \prec u' \end{array}}{B \vdash \texttt{field}\ f = t \diamond}\quad (\diamond\_\mathrm{Val})$$

Program w.f.

$$\frac{\vdash \overline{D}\ \diamond \qquad \texttt{Root} \vdash t \diamond}{\vdash \overline{A}\,\overline{a}\,\overline{D}\ t \diamond}\quad (\diamond\_\mathrm{Prog})$$

Fig. 4. Well-formedness Relations

5.5 Alias Analysis

In Section 3.2 we have informally explained why the assignment of field `x` in class `Y` was valid in the given SCALA code example. We give here a formal proof that the corresponding field valuation in a SCALETTA version given below of that code is well-formed.

```
class A {
  field T: Root.this!Object;
  class X {
    field x: A.this.T;         // "A.this.T"  means "this@X.T"
  }
}

class B extends !A {
  field T = Root.this!String;  // "Root.this" means "this@B"
  class Y extends !X {         // "!X"        means "this!X"
    field x = "foo";
  }
}
```

Note that we assume the presence of a top-level class `String` and that `"foo"` is an instance of that class.

To show that the field valuation is valid we have to apply the rule ($\diamond$_Val) to prove the following judgment:

$$\texttt{Y} \vdash \texttt{field x} = \texttt{"foo"} \diamond$$

This rule has several premises. However, the main difficulty is to prove the judgment given below. This judgment denotes the intuition that within class Y it is known that T is equal to String. This is the most difficult part because it is this judgment that requires some alias analysis.

$$\texttt{Y} \vdash \texttt{this @X .T} \rightarrow^{*} \texttt{this @Y @B !String}$$

We prove first that in the context of class Y, this @X and this @Y denotes the same enclosing instance, i.e. that $\texttt{Y} \vdash \texttt{this @X} \rightarrow \texttt{this @Y}$.

$$(\rightarrow^{abs}\_\text{Outer})\dfrac{(\prec^{abs}\_\text{Ext})\dfrac{(<^{abs}\_\text{This})\dfrac{}{\texttt{Y} \vdash \texttt{this} < \texttt{Y}} \quad \texttt{Y}\ \text{extends}\ \texttt{this !X}}{\texttt{Y} \vdash \texttt{this} \prec \texttt{this @Y !X}}}{\texttt{Y} \vdash \texttt{this @X} \rightarrow \texttt{this @Y}}$$

Then we use this result to reduce the prefix of this $@X\ .T$.

$$(\rightarrow^{abs}\_\text{Context})\dfrac{\texttt{Y} \vdash \texttt{this @X} \rightarrow \texttt{this @Y}}{\texttt{Y} \vdash \texttt{this @X .T} \rightarrow \texttt{this @Y .T}}$$

Finally we make use of the value of the field T in class B.

$$(\rightarrow^{abs}\_\text{Field})\dfrac{(<^{abs}\_\text{Outer})\dfrac{\texttt{class Y inside B}}{\texttt{Y} \vdash \texttt{this @Y} < \texttt{B}} \quad \texttt{B}\ \text{declares}\ (\texttt{field T} = \texttt{this @B !String})}{\texttt{Y} \vdash \texttt{this @Y .T} \rightarrow \texttt{this @Y @B !String}}$$

6 Typing Abstract Inheritance

The choice of abstract inheritance, i.e. the possibility for a class to inherit an arbitrary term instead of just a class, implied until here simplifications in the syntax, semantics and typing rules. However, with abstract inheritance it becomes far more challenging to detect the accidental overriding of a type field or an outer field than in a language where the class hierarchy is statically known, like SCALA. In this section we illustrate by examples why it is not safe to allow overriding of type fields and outer fields.

The type system presented before does not take into account these two problems. One way of solving these problems is to add dynamic tests that check at runtime that an object does not define two values for a same field and that it has no two different enclosing objects corresponding to a same class. But if we want a safe statically typed calculus we need mechanisms to resolve these problems at compile time. So, in this section we also present our solutions to statically prevent these overridings.

6.1 Field Roles

In SCALETTA a field can conceptually play different roles depending on where it is used. We introduce some terminology for describing these roles: we call *template field* a field that is indented to be used in an `extends` clause, we call *type field* a field that is used in the bound of another field, and we call *value field* a field that is used as an object. Note that in SCALETTA all fields can simultaneously play all these roles, to simplify the formalization we treat all the fields uniformly because they share the same characteristic of being potentially redefined in a subclass.

In real object-oriented languages, fields (also called members) have a statically determined role. In SCALA there are type fields and value fields, but no template fields; a field is categorized to be a type or a value from the moment of its declaration through the use of different keywords.

In order to have a safe type system, we need to *forbid* the overriding of type fields, as shown by the example of the next section. And in order to have an expressive type system, we need to *allow* the type system to exploit the actual value of a type field. Our choice of treating uniformly the different field roles forces us to extend these constraints to all kinds of fields. For instance, it forces us to abandon the overriding of value fields, even if it is harmful, and it allows us to exploit the value of a value field at compile time, a thing that most compilers do not do.

6.2 Overriding of Type Fields

In the type system presented so far, the typing rule ($\rightarrow^{abs}$_Field) which approximates terms like $t\,.\,f$ by $[t/\mathtt{this}]u$ in presence of a field valuation $\mathtt{field}\ f = u$ implicitly assumes that $\mathtt{field}\ f = u$ is the only existing valuation for f inherited by t. Another valuation with a different value would introduce an inconsistency in the type system and would break type safety. But let us illustrate this problem with a "concrete" example.

The following SCALA program is clearly unsafe because in class `C` the string `"foo"` held by the field `x` is added to the integer 3.

```
abstract class A {
  type T;
  val x: T;
}

class B extends A {
  type T = String;
  val x = "foo";
}

class C extends B {
  type T = Int;
  val y: Int = x + 3;
}
```

However a naive type system would accept it because:

– in class B, the bound T of the field x resolves to String, so it is legal for x to hold the value "foo".
– similarly, in class C the bound T of the field x resolves to Int, so it is legal to consider x as an integer and add it to 3.

The problem comes from the fact that when type-checking class B, we make the implicit assumption that the value of T is equal to String in any instance of B which is not always the case if we allow the overriding of this type field in the subclass C.

In languages with a static class hierarchy it is easy to prevent overriding of type fields, by checking that there is no path in this hierarchy that contains two valuations for the same field, as the path C $\rightarrow$ B $\rightarrow$ A in our example that contains two valuations for the field T.

But let us have a look at the following SCALETTA code which is a variant of the previous example.

```
class A {
  field T;
  field x: T;
}

class B extends !A {
  field T = !String;
  field x = "foo";
}

field b: !A = !B;

class C extends b {
  field T = !Int;
  field y: !Int = x + 3;
}
```

Note that the only difference is that class C inherits now from an instance b of B instead of directly inheriting from B. For the purpose of our argumentation, we "hide" the exact value of b behind the bound !A.

Now, without extra mechanism the only way of detecting an overriding of the field T in class C is to use the information that b holds an instance of B. But what if this value is hidden, as in our example, or even worse if b is an abstract field?

```
field b: !A;
```

Then, clearly we need an additional mechanism to follow the absence or presence of a valuation for a field in a given term, in our example in the term b used as parent of the class C.

6.3 Holes

In SCALETTA a term can be an instance of a class that has abstract fields. We say that such a term is *incomplete* and we call the fields that have no valuation in the term the *holes* of the term. We propose to add an annotation to field definitions to specify the holes of the value contained by this field. The key idea is that holes become the only fields that can be overridden. In our example we would write:

```
field b: !A  misses {};
```

to specify that `b` is an instance of `A` with an empty set of holes. This way it becomes possible to reject the overriding of the field `T` in class `C`, because `T` is not a hole of `b`. More generally, a field definition

```
field f: t misses {f1,...,fn};
```

means that f holds a value that expands to t and for which valuations of fields $f_1, \ldots, f_n$ are missing.

Let us see an example where the hole annotation is not empty. In fact such annotations appear in the encoding of methods. In these case the holes will correspond to the method parameters. If the method has type parameters, they will corespond to type fields. As our previous discussion has shown, it is important to forbid the overriding of this kind of the fields.

For instance, encoding the following `add` method declaration in class `Int`

```
field add(that: !Int): !Int;
```

results in the following declarations[2]

```
class Int$add$def {
  field that: !Int;
  field result: !Int;
}
field add: !Int$add$def misses {that};
```

The last line means that the field `add` holds an instance of `Int$add$def` with a missing valuation for the field `that`. It allows a class extending `add` to provide a valuation for the field `that`. Such classes appear in the encoding of an application of the method. For instance, the method call

```
40.add(2)
```

is encoded as[3]

```
class Apply extends 40.add {
  field that = 2;
}
!Apply.result
```

[2] We use the special character $ as a normal character in the names of classes generated by the method encoding. It must not be confused with an operator of the calculus.

[3] For clarity reasons, we use the numbers 40 and 2 to denote objects built from the classes `Zero` and `Succ`.

In the encoding of an implementation for the method `add` we subclass the field `add` and provide a body, as for example in class `Zero`:

```
class Zero$add$val extends !Int$add$def {
  field result = that;
}
field add = !Zero$add$val
```

The above valuation for `add` is legal because `!Zero$add$val` conforms to the declared type of `add`: it is indeed an instance of `Int$Add$def` and its only missing field valuation is the one for `that`.

We conclude the informal presentation of holes with the special hole annotation `*` that works as a wildcard for holes. Holes are mainly useful for value fields and template fields: an empty set of holes for a value field will ensure that the held value is complete and can consequently be used as receiver object, for template fields, holes allow to avoid overriding conflicts. As we want a uniform treatment of fields, type fields must also receive a hole annotation, but we can not give an exhaustive list of holes for type fields. For instance, a type field `T` bounded by `!Object` could receive the value `!Int` in one subclass and the value `!List` in another. If classes `Int` and `List` are abstract, values `!Int` and `!List` have a priori unrelated holes. To handle this kind of cases where the set of holes is not predictable we use the annotation `*` that matches an arbitrary set of holes:

```
field T: !Object misses {*};
```

6.4 Formalizing Holes

To prevent multiple valuations for a field, we define a relation on terms that returns the list of fields for which the term inherits a definition but no valuation. We call these fields the *holes* of the term. The judgment

$$B \vdash \text{holes}(t) = H$$

express the fact that in the context of the class B the set of holes of term t is H.

With that relation it is easy to prevent multiple valuations simply by allowing a field valuation in a class only if the field is included in the holes of the class parent. In order to be able to define this relation on terms, we need to know for each field its list of holes. For this reason, we augment field types with a list of holes (see Figure 5). In addition to field names, this list may contain the symbol $*$ which means that the field can contain more holes than the ones explicitly given.

In addition to multiple valuations, the holes let us solve another problem related to field valuations, namely missing field valuations. Indeed, the typing rule ($\diamond$_Get) states that the term $t\,.\,f$ is well-formed iff t inherits the declaration of field f. This does not guarantee that t inherits a valuation for the field f. To prevent this, we use the holes information to forbid any selection on terms that have any hole. This for sure will prevent any selection of a field f on a value that inherits no valuation for f.

All the modified and newly introduced rules to handle holes are summarized in Figure 5. The accessors defs(C) and vals(C), used for instance in rule (holes_New), return respectively the set of field definitions and the set of field valuations directly contained in a given class C.

Syntax (modifications only)

$$\begin{array}{ll} \text{Types } T & ::= t \; \texttt{misses} \; H \\ \text{Holes } \; H & ::= \{\bar{l}\} \mid \{\bar{l}, *\} \end{array}$$

Hole Approximation

(holes_This)
$$\frac{}{B \vdash \text{holes}(\texttt{this}) = \{\}}$$

(holes_New)
$$\frac{\texttt{class}\; C \;\texttt{extends}\; p \;\{\, \overline{d} \,\} \quad B \vdash \text{holes}([t/\texttt{this}]p) = H}{B \vdash \text{holes}(t \,\texttt{!}\, C) = (H \cup \text{defs}(C)) \backslash \text{vals}(C)}$$

(holes_Get)
$$\frac{\texttt{field}\; f : t \;\texttt{misses}\; H \;\texttt{inside}\; C}{B \vdash \text{holes}(t \,.\, f) = H}$$

(holes_Red)
$$\frac{B \vdash t \rightarrow u \quad B \vdash \text{holes}(u) = H}{B \vdash \text{holes}(t) = H}$$

(holes_Droping)
$$\frac{B \vdash \text{holes}(t) = \{\overline{f}, *\} \quad \overline{f'} \subset \overline{f}}{B \vdash \text{holes}(t) = \{\overline{f'}, *\}}$$

(holes_Opening)
$$\frac{B \vdash \text{holes}(t) = \{\overline{f}\}}{B \vdash \text{holes}(t) = \{\overline{f}, *\}}$$

Path and Declaration Well-Formedness (modifications only)

($\diamond$_New)
$$\frac{B \vdash \text{holes}(t) = \{\} \qquad \ldots}{B \vdash t \,\texttt{!}\, C \diamond}$$

+ same addition to rules ($\diamond$_Get) and ($\diamond$_Outer)

...

($\diamond$_Val)
$$\frac{\begin{array}{c} \texttt{field}\; f : u \;\texttt{misses}\; H \;\texttt{inside}\; A \\ B \vdash \text{holes}(t) = H \\ \texttt{class}\; B \;\texttt{extends}\; p \;\{\, \overline{d} \,\} \\ B \vdash \text{holes}([\texttt{this}\, \texttt{@} B/\texttt{this}]p) = H' \\ f \in H' \cup \text{defs}(B) \\ \ldots \end{array}}{B \vdash \texttt{field}\; f = t \diamond}$$

Fig. 5. Hole resolution

6.5 Overriding of Outer Fields

We have already explained the parallel between outer fields and normal fields, saying that outer fields were a special kind of field that come implicitly with the declaration of a class and whose value cannot make reference to the current instance of this class. Like type fields, outer fields are resolved at compile time, and like type fields they cannot suffer arbitrary overriding.

We start with a theoretical argument for this last claim: the typing rule ($\rightarrow^{abs}$_Outer) that lets one reduce the term $t\,\texttt{@}C$ to the term u if t expands to a term of the form $u\,\texttt{!}C$ implicitly assumes that t does not expand to a term $u'\,\texttt{!}C$ where $u \neq u'$. If such other term u' exists, the danger is to choose one term at compile time and the other one at runtime. It would mean that the approximation made by the type-checker for $t\,\texttt{@}C$ was in fact wrong, which opens an obvious breach in the type safety.

We now illustrate the problem with a concrete example. The following SCALETTA example shows that it is generally not safe to allow overriding of outer fields. An outer field is overridden if some object has two different enclosing instances corresponding to the same class. The following example is a variant of the one presented in Section 6.2. It suggests also how mixins can be encoded in SCALETTA.

```
class M {
  field T: !Object;
  field s: !Object;
  class N extends s {
    field x: T;
  }
}

class M1 extends M { field T = !String; field s = !Object; }
class A extends !M1!N { field x = "foo"; }

class M2 extends M { field T = !Int; field s = !A; }
class B extends !M2!N { field y: !Int = x + 3; }
```

The program above is unsafe because in class `B` the string value `"foo"` held by the field `x` is added to the integer 3. However a naive type system would accept it because:

- in class `A`, the bound `this@N.T` of the field `x` resolves to `!String`, so it is legal for `x` to hold the value `"foo"`.
- similarly, in class `B` the bound `this@N.T` of the field `x` resolves to `!Int`, so it is legal to consider `x` as an integer and add it to `3`.

The problem is that class `B` has two incompatible enclosing instances corresponding to the class `N`, namely `!M1` and `!M2`. More precisely we have the following expansion chain:

$$\texttt{!B} \prec \texttt{!M2!N} \prec \texttt{!M2.s} = \texttt{!A} \prec \texttt{!M1!N}$$

To break the inheritance cycle, we need a way to forbid class `N` to inherit an instance of itself. But the parent `s` of `N` is abstract, so we need to attach a field annotation to the declaration of `s` to specify that it cannot hold an instance of `N`. We call such a mechanism *class exclusion* and formalize it in the next section. The syntax for such annotations is:

```
def s: !Object excludes {N};
```

With this definition a field valuation for `s` may not contain an instance of `N`. The field valuation `field s = !A` in class `M2` becomes illegal because `!A` is clearly an instance of `N` and the whole unsafe program gets rejected as expected.

6.6 Formalizing Class Exclusion

The problem of multiple enclosing instances arises if a value inherits different instances of a same class. Therefore, to avoid the problem, we make it impossible for a value to inherit more than one instance of a given class. To do that, we define a relation that associates to a term a list of classes of which the term inherits no instance. This relation is then used to make sure that the parent of a class does not already inherit an instance of that class. The definition of this class exclusion relation requires an additional annotation for field types, namely a list of classes of which the field is not an instance.

The judgment

$$B \vdash \text{notInst}(t, \overline{C})$$

expresses the fact that in the context of the class B the term t is not an instance of any of the classes $\overline{C}$. Note that for extensibility reasons the list of classes $\overline{C}$ is generally not exhaustive.

The Figure 6 summarizes the modified and the newly introduced rules to handle class exclusion.

The basic rules for class exclusion are already powerful enough to avoid the inheritance of a mixin by itself, as in our example. In the next section we show the limitations of these rules and suggest a natural extension.

6.7 Group Exclusion

The idea of class exclusion is very simple but unfortunately it does not cover all interesting uses of abstract inheritance; for type-checking the encoding of a method call we need to extend the mechanism of class exclusion with the concept of *group of classes*. Remind that a method call $e.m(3)$ is encoded as

```
class Apply extends e.m {
  field arg = 3
}
!Apply.result
```

For this class declaration to be well-formed, we must prove that `e.m` is not already an instance of `Apply`. So at the moment when we define the method `m` we can just write

```
field m: M$def misses {arg} excludes {Apply}
```

to specify that `m` can only hold values that are not instance of `Apply`. If there are more than one call to `m` in the program we can extend the class exclusion annotation accordingly:

Syntax (modifications only)

$$\text{Types } T ::= t \; \texttt{misses} \; H \; \texttt{excludes} \; \overline{C}$$

Class Exclusion Approximation

(notInst_Red)
$$\frac{B \vdash t \rightarrow u \qquad B \vdash \text{notInst}(u, C)}{B \vdash \text{notInst}(t, C)}$$

(notInst_Field)
$$\frac{\texttt{field} \; f : t \; \texttt{misses} \; H \; \texttt{excludes} \; \overline{C} \; \texttt{inside} \; C}{B \vdash \text{notInst}(t \,.\, f, C_i)}$$

(notInst_New)
$$\frac{C \text{ extends } p \qquad B \vdash \text{notInst}([t/\texttt{this}]p, C') \qquad C' \neq C}{B \vdash \text{notInst}(t \,\texttt{!} C, C')}$$

Declaration Well-Formedness (modifications only)

($\diamond$_Class)
$$\frac{\texttt{class} \; C \; \texttt{inside} \; B \qquad p = t \Rightarrow B \vdash \text{notInst}(t, C) \qquad \ldots}{\vdash \texttt{class} \; C \; \texttt{extends} \; p \, \{ \, \overline{d} \, \} \diamond}$$

($\diamond$_Val)
$$\frac{\begin{array}{c}\texttt{field} \; f : u \; \texttt{misses} \; H \; \texttt{excludes} \; \overline{C} \; \texttt{inside} \; C \\ \forall i.\; B \vdash \text{notInst}(t, C_i) \qquad \ldots\end{array}}{B \vdash \texttt{field} \; f = t \diamond}$$

Fig. 6. Class exclusion annotations

```
field m: M$def misses {arg} excludes {Apply1, ..., Applyn}
```

The problem with such a schema is that it is not extensible, in the sense that the number of possible application will necessary be bounded. And we do not want to restrict the use of a method to a particular number of applications because it precludes the idea of separate compilation. So a simple idea is to group classes and to exclude groups instead of classes.

In this case we declare a group called `ApplyGroup`

```
group ApplyGroup;
```

And we let the field `m` exclude this group:

```
field m: M$def misses {arg} excludes {ApplyGroup}
```

Now when declaring a class `Apply`, we link it to the group `ApplyGroup`.

```
class Apply in ApplyGroup extends e.m {
  field arg = 3
}
!Apply.result
```

The key idea is that if `m` excludes the group `ApplyGroup`, it will a fortiori exclude the class `Apply` which is a member of this group.

The adding of groups looks as a natural and harmful extension of our mechanism of class exclusion. But rather than formalizing groups directly we prefer simulate them with classes, which has the advantage to avoid enriching unnecessarily the syntax with new concepts. We consider that each class defines implicitly a group and that a class is member of a group if it inherits from the corresponding class. This idea is formalized by adding a new rule to the class exclusion relation:

$$\text{(notInst\_Group)}\ \frac{\begin{array}{c} B \vdash \text{notInst}(t,\, C) \\ \texttt{class}\ C'\ \texttt{extends}\ u\ \{\,\overline{d}\,\} \quad \texttt{class}\ C'\ \texttt{inside}\ B' \\ B' \vdash u\ <\ C \end{array}}{B \vdash \text{notInst}(t,\, C')}$$

Now, to make all methods member of a group `Apply`, we first define a "mixin" `Apply`:

```
class MixApply {
  def s: !Object excludes {Apply};
  class Apply extends s;
}
```

And we declare all fields corresponding to methods to exclude the group `Apply`.

```
def m: M$def misses {arg} excludes {Apply};
```

Now, when calling a method, we first apply the mixin:

```
class MixApply1 extends MixApply { field s = e.m; }
class Apply1 extends MixApply1!Apply {
  field arg = 3
}
!Apply1.result
```

7 Undecidability and Typing Strategies

Typing in the field of programming languages can be seen as a way of approximating the result of an expression without evaluating it. Evaluating an expression consists of replacing abstractions by their value. So a good reason not to evaluate an expression at compile time is that some abstractions have no value. Even when abstractions have values, typing algorithms usually prefer to approximate the abstraction using its declared interface (or type), instead of its value, in order to prevent the compiler from looping (types usually do not loop). From this point of view typing can be seen as a kind of abstract evaluation. SCALETTA makes this close link between typing and evaluation completely explicit because the typing relations generalize the evaluation relations, as explained in Section 5.

It follows that typing is undecidable because it can happen that the only way of checking that an expression conforms to a bound is to evaluate it. A naive solution to "approach decidability" is to never use the value of a field during type-checking but always use its bound. But if we do that we can not simulate type fields with term fields anymore because the main property of type fields is precisely that their value is used at compile time. There is clearly a trade-off between the need of making the type-checking "more decidable", i.e. that it terminates on more programs, and the need of being able to write typed solutions to interesting problems.

Having identified that a source of undecidability in the typing of SCALETTA is an excessive usage of field valuations, we define a *typing strategy* as a policy that restricts this usage. From a theoretical point of view, such a policy controls the usage of the rule ($\rightarrow^{abs}$_Field). It is important to note that a proof of type-safety for our type system would still hold if the usage of the deduction rules was restricted by a typing strategy. A type-checker using a restrictive typing strategy would just accept less programs than one using a more liberal strategy, but all accepted programs would still be guaranteed not to cause errors at runtime.

In conclusion we have a type system with a degree of decidability and expressiveness which is parameterizable by a typing strategy. We considered two simple typing strategies: one that requires that the programmer explicitly annotates field valuations that can be followed by the type-checker and another that infers from the context this right, for instance if a field plays the role of a type field. In our compiler we adopted the second strategy because it does not require additional annotations from the programmer and because there are in principle cases where a same field could be considered as a type or as a value in different contexts. Our experience with our interpreter seems to indicate that the choosen strategy works well; it accepts non-trivial programs, signals useful error messages and does not unexpectedly loop.

8 Globally Unique Names

Our calculus requires that all declared classes have a globally unique name. This seems a rather strong restriction, but we show here that this is not more restrictive than what exists in other programming languages.

In Java, classes declared in different scopes may have the same name. However, the declared name of a class is only what is called its *short name*. A class has also a *full name* which is built by prefixing its short name with the short names of all it enclosing classes and the name of its package. This full name has to be globally unique. So, even a mainstream language like Java has the same kind of restriction as our calculus.

In fact, one should distinguish between the identity of an entity and the name of that entity. In any language, each entity (class, field, variable, ...) has its own unique identity. Often, it has also a name, but that is just an attribute of that entity. In many languages, where code is just a string of characters, an entity may only be referenced through its name. In order to be more programmer-friendly, most of these languages allow the reuse of the same names in different scopes. These languages define also a set of rules which given a name and a scope determine which entity is referenced by that name. This process is called *name resolution*. Once name resolution has been applied, every name is linked to a particular entity. At that point, all entities can be alpha-renamed to obtain a program that exhibits the same behavior but where all entities have globally unique names.

The requirement for globally unique names in our calculus simply means that we consider only programs for which name resolution has already been performed. This means that we do not need to define rules that regulate name resolution. That is an advantage because in this paper we are concerned by semantics and typing issues, not name resolution issues.

We acknowledge that name resolution is not an easy problem, especially in languages like Java where it relies on types and has to be performed in parallel with type-checking. However, a bit like syntactic sugar, name resolution is just a way to make programming more friendly. It does not contribute to the expressive power of the language. For example, Java without import statements would still be the same language (same semantics and typing rules), but it would just be a bit less programmer-friendly as classes would have to be referenced through their full name. Separating conceptually name resolution and type analysis is also similar to separating conceptually type inference and typing.

In our calculus, it pays off to have no name resolution rules because our semantics and typing rules do not involve the duplication of name binders. Thus, assuming that names are globally unique effectively simplifies the calculus. It has also the advantage that any entity can be referenced from any point of the program. In calculi of the λ-calculus family, there are usually many rules that duplicate name binders and thus create different entities with the same name. Thus, starting with globally unique names would not bring much and that is why those calculi usually have no such restriction. However, to avoid name captures, their semantics and typing rules usually use, now well-known but still non-trivial, alpha-renaming rules. Our calculus does not require such rules.

After explaining why it is justified to conceptually separate name resolution and typing in Scaletta, we describe briefly how these two mechanisms are closely inter-linked in our implementation of the calculus. The problem is always the same: there is an occurrence of a field name f in the program and we want to link it with a field declaration. When we fail to do it, we reject the program as ill-typed, when we succeed we can exploit the declared type attached to the found declaration in the rest of the analysis. There are two cases: either the field name f starts a term, either it is selected on a non-empty term as in $t\,.\,f$. We say a declaration of a field f is a *member* of a term t, if t is an instance of a class C that contains a declaration for f. For a field name f starting a term in a class C, it is first searched if f is a member of the current instance of C, otherwise the search is recursively launched from the enclosing class C' of C. For a remote selection $t\,.\,f$, f must necessary be a member of t. Note that f will never be interpreted as a member of an enclosing instance of t.

9 Related Work

Both concepts, inner classes and virtual types, have been studied and well understood separately. Inner classes are described in [3] and virtual types are formalized in [4].

The interaction between inner classes and virtual types is less well understood. We know only about four works related to this problem.

In [9] the authors aim at formalizing the type system of the Scala programming language. They do not especially focus on inner classes and virtual types. They describe a calculus that also includes other concepts like object identity, mixins and first-class class constructors. Its type system is proved to be sound. However the calculus is rather complex and therefore is very hard to use as a basis to understand the interaction between inner classes and virtual types. We claim to have a simpler calculus with just the required concepts to formalize inner classes and virtual types.

Beta is an object-oriented programming language with inner classes and virtual classes. Virtual classes in Beta are equivalent to the notion of virtual types described in our paper. In [10] the author considers the problem of typing virtual classes in the presence of inner classes in Beta. The main difference between his and our work is that he describes the algorithms used by the Beta compiler to perform the semantic analysis of a program whereas we give a calculus with a type system that formally defines well-formed programs. We do not describe in the paper an algorithm to build well-formedness proofs. However we have written an interpreter which incorporates such algorithms. The algorithms presented for Beta performs simultaneously name and type analysis. A contribution of our work is to show that it is possible to specify the static semantics without having to include rules to perform name analysis. Note however that our interpreter implements a name analysis and thus does not impose globally unique names. The same paper contains interesting examples describing more intricate situations than our examples.

Gbeta [11] is an extension of Beta with the possibility to extend virtual classes. This mechanism is similar to our notion of abstract inheritance. Our understanding of Gbeta is very weak, and we would be interested to know if our solutions for typing abstract inheritance could somehow be used to type virtual classes in Gbeta.

We lately discovered in [12, 13] that the author had developed ideas and solutions for a calculus of classes and objects very close to the ones we developed for Scaletta. The author distinguishes in his work two approaches to inheritance: the modificationist approach that views classes mainly as code libraries whereas the specialisationist approach insists on considering them as abstractions. As we do, he recommends the later, claiming that overriding is not essential. He develops also a mechanism similar to holes and uses it in exactly the same way as we do for encoding methods. However he is more extremist than we are when he claims that only one kind of names is necessary. On the contrary we think after our experience that it is important to make a clear distinction between class names and field names and that any unification of both concepts is likely artificial. His work goes beyond the work presented in this paper when he tries to integrate in his formalism a kind of structural subtyping for handling full polymorphism. Finally, the author is also missing a soundness proof that would validate his interesting ideas.

10 Conclusion and Future Work

The main contribution of this work is a better understanding of the interaction between inner classes and virtual types through a simple and novel calculus of classes and objects. We describe formally the non-trivial mechanism needed to type virtual types in the presence of inner classes. We explicit the required alias analysis and show that it requires only a few simple rules provided that each inner class comes with an implicit and accessible outer field.

This work has application for the design of type checkers for languages that combine inner classes and virtual types. It can also be used to understand type systems of already existing programming languages that combine these two aspects like BETA or SCALA. It may even be used to understand, at least partially, languages with inner classes and parameterized classes like JAVA 5.0 which require the same alias analysis as the one formalized in our calculus.

Another contribution of this work is our calculus. It can be seen as an intermediate language for object oriented languages because it has no notion of binder and does not use alpha-renaming, but still remains perfectly appropriate to support a powerful type system. The unification of terms and types lets us define typing as abstract evaluation of the program. We are not aware of another similar result in the field of oriented programming languages. Finally we hope our solutions for the typing of abstract inheritance can be of interest for people working on the difficult problem of typing virtual classes.

Our priority in the future is to prove the soundness of our type system. We are also interested in designing a variant that includes the concepts of object identity and state. The main implication of such a change would be the introduction of the notion of stable term which describes a term whose evaluation always return the same object reference.

Acknowledgments

The authors are grateful to the following people whose comments on previous drafts of this paper helped improving its readability: Martin Odersky, Michel Schinz, Burak Emir, Iulian Dragos, Stéphane Micheloud, Sébastien Briais and Daniel Bünzli. We thank also the anonymous reviewers of ESOP05 and ECOOP05 for taking the time to read and comment this work.

References

1. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: The BETA programming language. In Shriver, B., Wegner, P., eds.: Research Directions in Object-Oriented Programming, Cambridge, MA (1987)
2. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the scala programming language. Technical report IC/2004/64, EPFL, Switzerland (2004)
3. Igarashi, A., Pierce, B.C.: On inner classes. In: European Conference on Object-Oriented Programming (ECOOP). (2000) Also in informal proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL). Full version in *Information and Computation*.
4. Igarashi, A., Pierce, B.C.: Foundations for virtual types. (1999) Also in informal proceedings of the *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1999. Full version in *Information and Computation*, 175(1): 34–49, May 2002.
5. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA). (1999) Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.
6. Altherr, P., Cremet, V.: Scaletta web page. http://lamp.epfl.ch/∼paltherr/scaletta (2004)
7. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Java Series, Sun Microsystems (1996) ISBN 0-201-63451-1.
8. Bruce, K.B., Odersky, M., Wadler, P.: A statically safe alternative to virtual types. In: ecoop98. Volume 1445., Brussels, Belgium (1998) 523–549
9. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Proceedings of the European Conference on Object-Oriented Programming, Darmstadt, Germany (2003)
10. Madsen, O.L.: Semantic analysis of virtual classes and nested classes. In: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (1999) 114–131
11. Ernst, E.: gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark (1999)
12. Torgersen, M.: Inheritance is specialization. In: The Inheritance Workshop, with ECOOP 2002. (2002) `http://www.cs.auc.dk/~eernst/inhws/`.
13. Torgersen, M.: Unifying Abstractions. PhD thesis, Computer Science Department, University of Aarhus (2001)