A Core Calculus for Scala Type Checking

Martin Odersky, EPFL
joint work with François Garillot, Vincent Cremet, and Serguei Leunglet.

Invited Talk,
Mathematical Foundations of Computer Science (MFCS), Štúr Lesta,
Slovakia, August 31, 2006.

Example: Peano Numbers

To give a feel for the language, here’s a Scala implementation of
natural numbers that does not resort to a primitive number type.

```scala
trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat = new Nat(this);
  def + (x: Nat): Nat = if (isZero) this else succ + x.pred;
  def - (x: Nat): Nat = if (isZero) this else pred - x.succ;
}
```

Here are the two canonical implementations of Nat:

```scala
case class Succ(n: Nat) extends Nat {
  def isZero: Boolean = false;
  def pred: Nat = n
}
```

```scala
case class Zero extends Nat {
  def isZero: Boolean = true
  def pred: Nat = throw new Error("Zero.pred");
}
```

No Statics!

A component should refer to other components not by hard links,
but only through its required interfaces.

Another way of expressing this is:

> All references of a component to others should be via its
members or parameters.

In particular, there should be no global static data or methods that
are directly accessed by other components.

This principle is not new.
But it is surprisingly difficult to achieve, in particular when we
extend it to classes.

Scala

Scala is an object-oriented and functional language which is
completely interoperable with Java and .NET.

It removes some of the more arcane constructs of these
environments and adds instead:

1. a uniform object model,
2. pattern matching and higher-order functions,
3. new ways to abstract and compose programs.

Components

Scala provides new ways to build component systems.

A component is a program part, to be combined with other parts in
larger applications.

Requirement: Components should be reusable.

To be reusable in new contexts, a component needs interfaces
describing its provided as well as its required services.

Most current components are not very reusable.
Most current languages can specify only provided services, not
required services.

Note: Component ≠ API !

Functors

One established language abstraction for components are SML
functors.

Here,

| Component | ≅ | Functor or Structure |
| Interface | ≅ | Signature |
| Required Component | ≅ | Functor Parameter |
| Composition | ≅ | Functor Application |

Sub-components are identified via sharing constraints.

Shortcomings:

- No recursive references between components
- No inheritance with overiding
- Structures are not first class.
Modules are Objects

In Scala:

- \textit{Component} \triangleq \textit{Class}
- \textit{Interface} \triangleq \textit{Abstract Class}, or \textit{Trait}
- \textit{Required Component} \triangleq \textit{Abstract Member} or “Self”
- \textit{Composition} \triangleq \textit{Modular Mixin Composition}

Advantages

- Components instantiate to objects, which are first-class values.
- Recursive references between components are supported.
- Inheritance with overriding is supported.
- Subcomponents are identified by name
  \Rightarrow no explicit “wiring” is needed.

Abstract Types

Here is a type of “cells” using object-oriented abstraction.

```
trait AbsCell {
  type T;
  val init : T
  private var value : T = init
  def get : T = value
  def set(x : T) : Unit = { value = x }
}
```

The AbsCell class has an abstract type member \( T \) and an abstract value member \( \text{init} \). Instances of that class can be created by implementing these abstract members with concrete definitions.

```
val cell = new AbsCell { type T = Int; val init = 1 }
cell.set(cell.get + 2)
```

The type of \( \text{cell} \) is \( \text{AbsCell} \{ \text{type } T = \text{Int} \} \).

Language Constructs for Components

Scala has three concepts which are particularly interesting in component systems:

- \textit{Abstract type members} allow to abstract over types that are members of objects.
- \textit{Self-type annotations} allow to abstract over the type of “self”.
- \textit{Modular mixin composition} provides a flexible way to compose components and component types.

Theoretical foundations: \( \nu \text{Obj} \) calculus [Odersky et al., ECOOP'03], Featherweight Scala [this conference].

Scala’s concepts subsume SML modules.

More precisely, (generative) SML modules can be encoded in \( \nu \text{Obj} \), but not vice versa.

Path-dependent Types

It is also possible to access \( \text{AbsCell} \) without knowing the binding of its type member.

For instance:

```
def reset(c : AbsCell) : Unit = c.set(c.get + 2)
```

Why does this work?

- \( c.\text{init} \) has type \( c.T \)
- The method \( \text{reset} \) has type \( c.T \rightarrow \text{Unit} \)
- So the formal parameter type and the argument type coincides.

\( c.T \) is an instance of a \textit{path-dependent types}.

```
In general, such a type has the form \( x_1, \ldots, x_n, \ell \) where:
- \( x_1 \) is an immutable value
- \( x_2, \ldots, x_n \), are immutable fields, and
- \( \ell \) is a type member of \( x_n \).
```

Safety Requirement

Path-dependent types rely on the immutability of the prefix path.

Here is an example where immutability is violated.

```
val flip = false
def f(c : AbsCell) = {
  flip = flip
  if (flip) new AbsCell { type T = Int; val init = 1 } 
  else new AbsCell { type T = String; val init = "" }
  c.set(f(c).get) // illegal!
}
```

Scala’s type system does not admit the last statement, because the computed type of \( f(c).\text{get} \) would be \( f(c).T \).

This type is not well-formed, since the method call \( f(c) \) is not a path.

Foundations

- A language like Scala is complicated.
- How do we know we have the right design?
- How can we convince ourselves that types are sound and can be computed?
- We would like to have a small calculus which captures the “essence” of Scala, in particular the things which are relatively new.
The νObj Calculus

νObj [ECOOP 2003] is a calculus for a Scala-like language. It contains a nominal (i.e. declaration-based) type system with:
- abstract types,
- mixin composition,
- nested classes,
- explicit self types.
It also contains a construct not present in Scala: first-class classes, i.e. classes may be treated as other values.
This calculus can encode FS_2.
For that reason, typechecking in νObj is known to be undecidable.

FS: Syntax

<table>
<thead>
<tr>
<th>Alphabet</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha ), ( \nu )</td>
<td>Variable</td>
</tr>
<tr>
<td>( \tau )</td>
<td>Value label</td>
</tr>
<tr>
<td>( A )</td>
<td>Type label</td>
</tr>
<tr>
<td>( M, N )</td>
<td>Member dec</td>
</tr>
<tr>
<td>( T )</td>
<td>Field decl / def</td>
</tr>
<tr>
<td>( t )</td>
<td>Method decl / def</td>
</tr>
<tr>
<td>( x )</td>
<td>Class def</td>
</tr>
<tr>
<td>( s, t, u )</td>
<td>Trait extends ( T { \nu } )</td>
</tr>
<tr>
<td>( p )</td>
<td>Variable</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>Field selection</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>Method call</td>
</tr>
<tr>
<td>( \tau { \nu } )</td>
<td>Object creation</td>
</tr>
<tr>
<td>( S, T, U )</td>
<td>Type selection</td>
</tr>
<tr>
<td>( p : A )</td>
<td>Singleton type</td>
</tr>
<tr>
<td>( T { \nu } )</td>
<td>Type signature</td>
</tr>
</tbody>
</table>

Example: Peano Numbers revisited

```scala
 tn0 extends Nat { th0 } | defSZero() : Boolean |
defSucc() : Nat { th1 |
defisZero() : Boolean = false |
defisPred() : Nat = th0 |

defsuc0() : Nat = ( val result = new th0Succ() result ) |
defsuc1() : Nat = |
if(th0Succ()) th1 |
defsuc2() : Nat = |
if(th1Succ()) th2 |
defsuc3() : Nat = |
if(th2Succ()) th3 |
val zero = new th0 |
defisZero() : Boolean = true |
defisPred() : Nat = error("zeropred") |
```

Example: Generic Lists

```scala
tnList extends Any { th0 |
tnListElem = List { th0 | type Elem = th0Elem |
defisEmpty() : Boolean |
defisHead() : th0Elem |
defisTail() : thisedListElem |
} |
tnNil extends List { th0 |
defisNil() : Boolean = true |
defisHead() = |
defisTail() = |
defisZero() : Boolean = false |
defisSucc() : Boolean = true |
defisPred() : Nat = |
val hd : th0Elem = |
val tl : thisedListElem = |
val nil : th0Elem = |
val cons : th0Elem = |
defisZero() : Boolean = false |
defisSucc() : Boolean = true |
defisPred() : Nat = |

val cons : th0Elem = |
defisZero() : Boolean = false |
defisSucc() : Boolean = true |
defisPred() : Nat = |

val list2 = new Cons { th0 |
type Elem = Nat |
val hd : Nat = zeroSuc0(null) |
val tl : thisedListElem = |
val cons : th0Elem = |
defisZero() : Boolean = false |
defisSucc() : Boolean = true |
defisPred() : Nat = |
```

Type Assignment

- \( x : T \in \Gamma \) (package)
- \( \Gamma \vdash T : T \leftarrow \Theta \) (DEFINITION)
- \( \Gamma \vdash S \leftarrow \Theta \) (DEFINITION)
- \( \Gamma \vdash a : (x : T) : U = \) (SELECT)
- \( \Gamma \vdash p : T \leftarrow \Theta \) (singleton)
- \( \Gamma \vdash p : p : type \)
See Paper for ...

- Judgements for member subtyping $\ll$ and well-formedness $\mbox{WF}$.
- An operational semantics.
- An algorithmic formulation of the calculus, with the following differences:
  - Some judgement forms have been split.
  - Transitivity has been eliminated in the subtyping rules.
  - A notion of used definitions was added to the rules which act as locks to prevent cycles in typing derivations.
  - A proof of the decidability of typing and subtyping in Algorithmic FS

Future Work

1. Soundness proof for operational semantics (hopefully finished soon)
2. The lock-free version of the calculus is more expressive than the algorithmic one. There are programs that typecheck lock-free but fail due to a cycle in the locking version.
   - Can we refine locks so that the two versions become equivalent?
3. Extensions of the calculus, with
   - Polymorphic methods
   - Type bounds
   - Abstract inheritance/higher-order polymorphism
4. A call-by-value version of the calculus

Relationship between Scala and Other Languages

Main influences on the Scala design:
- Java, C# for their syntax, basic types, and class libraries,
- Smalltalk for its uniform object model,
- Krr for systematic naming,
- ML, Haskell for many of the functional aspects,
- O'Caml, Haskell, PLT-Scheme, as other combinations of FP and OOP.
- Pizza, MultiJava, Nice as other extensions of Java with functional ideas.

(Too many influences in details to list them all)
Scala also seems to influence other new language designs, see for instance the closures and comprehensions in C# 3.0.

Related Language Research

Mixin composition: Blache (linear), Duggan, Hirschkorn (mixin-modules), Sanner et al. (traits), Platt et al. (units, Jazza), Zenger (Keris).

Abstract type members: Even more powerful are virtual classes (Cook, Erns, Odersman)

Explicit self-types: Vuillon and Rémy (O'Caml)
Conclusion

- Despite 10+ years of research, there are still interesting things to be discovered at the intersection of functional and object-oriented programming.
- Much previous research concentrated on simulating some of X in Y, where \( X, Y \in \{ \text{FP}, \text{OOP} \} \).
- More things remain to be discovered if we look at symmetric combinations.
- Scala is one attempt to do so.

Try it out: scala-zepflch

Thanks to the (past and present) members of the Scala team:
Philiipe Allberg, Vincent Cremer, Julian Dangre, Gilles Dubochet,
Barak Emir, Sean McDermid, Stéphane Micheloud, Nickay
Mihaylov, Michel Schikorski, Lee Scoons, Erik Steenman, Matthias Zenger.