A Core Calculus for Scala Type Checking

Martin Odersky, EPFL

joint work with Francois Garillot, Vincent Cremet, and Sergueï Lenglet.

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

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) novel ways to *abstract* and *compose* programs.

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.

```
trait Nat {
  def isZero: boolean;
  def pred: Nat;
  def succ: Nat = new Succ(this);
  def + (x: Nat): Nat = if (x.isZero) this else succ + x.pred;
  def − (x: Nat): Nat = if (x.isZero) this else pred − x.pred;
}
```

Here are the two canonical implementations of Nat:

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

```
object Zero extends Nat {
  def isZero: boolean = true;
  def pred: Nat =
    throw new Error("Zero.pred");
}
```

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 $\neq$ API !

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.

Functors

One established language abstraction for components are SML functors.

Here,

Component	$\hat{=}$	*Functor* or *Structure*
Interface	$\hat{=}$	*Signature*
Required Component	$\hat{=}$	*Functor Parameter*
Composition	$\hat{=}$	*Functor Application*

Sub-components are identified via sharing constraints.

Shortcomings:

- No recursive references between components
- No inheritance with overriding
- Structures are not first class.

Modules are Objects

In Scala:

Component	$\hat{=}$	*Class*
Interface	$\hat{=}$	*Abstract Class, or Trait*
Required Component	$\hat{=}$	*Abstract Member* or *"Self"*
Composition	$\hat{=}$	*Modular Mixin Composition*

Advantages:

- Components instantiate to objects, which are first-class values.
- Recursive references between components are supported.
- Inheritance with overriding is supported.
- Sub-components are identified by name

  $\Rightarrow$ no explicit "wiring" is needed.

Language Constructs for Components

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

- *Abstract type members* allow to abstract over types that are members of objects.
- *Self-type annotations* allow to abstract over the type of "self".
- *Modular mixin composition* provides a flexible way to compose components and component types.

Theoretical foundations: νObj calculus [Odersky et al., ECOOP03], Featherweight Scala [this conference].

Scala's concepts subsume SML modules.

More precisely, (generative) SML modules can be encoded in νObj, but not *vice versa*.

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 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 cell is AbsCell { **type** T = int }.

Path-dependent Types

It is also possible to access AbsCell without knowing the binding of its type member.

For instance: **def** reset(c : AbsCell): unit = c.set(c.init);

Why does this work?

- c.init has type c.T
- The method c.set has type c.T ⇒ unit.
- So the formal parameter type and the argument type coincide.

c.T is an instance of a *path-dependent* type.

In general, such a type has the form $x_0.; \ldots ; .x_n.t$, where

- x_0 is an immutable value
- $x_1, \ldots, x_n$ are immutable fields, and
- t 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.

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

Scala's type system does not admit the last statement, because the computed type of f().get would be f().T.

This type is not well-formed, since the method call f() 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 $F_{<:}$.

For that reason, type-checking in νObj is known to be undecidable.

Featherweight Scala

The undecidability result for νObj relies on first-class classes.

These are absent in Scala, so Scala type checking might still be decidable!

To explore these issues, we studied a new system: FS, for Featherweight Scala.

Featherweight Scala is designed to be a minimal subset of Scala that still captures its essence.

Featherweight Scala is a subset of real Scala: Every FS program is also a legal Scala program.

FS: Syntax

Alphabets	x, y, z, φ			Variable
	a			Value label
	A			Type label
Member decl	M, N	$::=$	$\mathsf{val}\, a : T \;= t$	Field decl / def
			$\mathsf{def}\, a\, (\overline{y : S}) : T \;= t$	Method decl / def
			$\mathsf{type}\, A \;= T$	Type decl / def
			$\mathsf{trait}\, A\, \mathsf{extends}\, \overline{T}\, \{\varphi \mid \overline{M}\}$	Class def
Term	s, t, u	$::=$	x	Variable
			$t.a$	Field selection
			$s.a\, (\overline{t})$	Method call
			$\mathsf{val}\, x = \mathsf{new}\, T\, ; t$	Object creation
Path	p	$::=$	$x \;\mid\; p.a$	
Type	S, T, U	$::=$	$p.A$	Type selection
			$p.\mathbf{type}$	Singleton type
			$\overline{T}\, \{\varphi \mid \overline{M}\}$	Type signature

Example: Peano Numbers revisited

```
trait Nat extends { this0 |
    def isZero(): Boolean
    def pred(): Nat
    trait Succ extends Nat { this1 |
        def isZero(): Boolean = false
        def pred(): Nat = this0
    }
    def succ(): Nat = ( val result = new this0.Succ; result )
    def +(other : Nat): Nat =
        if (this0.isZero()) other else this0.pred().+(other.succ())
    def −(other : Nat): Nat =
        if (other.isZero()) this0 else this0.pred().−(other.pred())
}
val zero = new Nat { this0 |
    def isZero(): Boolean = true
    def pred(): Nat = error("zero.pred")
}
```

Example: Generic Lists

```
trait List extends Any { this0 |
    type Elem
    type ListOfElem = List { this1 | type Elem = this0.Elem }
    def isEmpty(): Boolean
    def head(): this0.Elem
    def tail(): this0.ListOfElem
}

trait Nil extends List { this0 |
    def isEmpty(): Boolean = true
    def head(): this0.Elem =
        error("Nil.head")
    def tail(): this0.ListOfElem =
        error("Nil.tail")
}

trait Cons extends List { this0 |
  val hd : this0.Elem
  val tl : this0.ListOfElem
  def isEmpty(): Boolean = false
  def head(): this0.Elem = hd
    def tail(): this0.ListOfElem = tl
}

val list2 = new Cons { this0 |      // List(2)
    type Elem = Nat
    val hd : Nat = zero.succ().succ()
    val tl : this0.ListOfElem =  new Nil { type Elem = Nat } }
```

Type Assignment

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \ (\textsc{path-var})$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S \ni \mathsf{val}\, a : T \ = u}{\Gamma \vdash t.a : T} \ (\textsc{select})$$

$$\frac{\Gamma \vdash p : T}{\Gamma \vdash p : p.\mathbf{type}} \ (\textsc{singleton})$$

$$\frac{\begin{array}{c}\Gamma \vdash s : S \\ \Gamma \vdash \bar{t} : \overline{T'} \qquad \Gamma \vdash \overline{T'} <: \overline{T} \\ \Gamma \vdash S \ni \mathsf{def}\, a \left(\overline{x : T}\right) : U \ = u\end{array}}{\Gamma \vdash s.a\left(\bar{t}\right) : U} \ (\textsc{method})$$

$$\frac{\begin{array}{cc}\Gamma, x : T \vdash t : S & x \notin \mathrm{fn}(S) \\ \Gamma \vdash T \prec_{\varphi} \overline{M_c} & \Gamma \vdash T \ \mathrm{WF}\end{array}}{\Gamma \vdash \mathsf{val}\, x = \mathsf{new}\, T\,; t : S} \ (\textsc{new})$$

Expansion and Membership

$$\frac{\forall i,\ \Gamma \vdash T_i \prec_\varphi \overline{N_i}}{\Gamma \vdash \overline{T}\,\{\varphi \mid \overline{M}\} \prec_\varphi \left(\biguplus_i \overline{N_i}\right) \uplus \overline{M}}\ (\prec\text{-SIGNATURE})$$

$$\frac{\Gamma \vdash p : T \qquad \Gamma \vdash T \prec_\varphi \overline{M}}{\Gamma \vdash p.\mathbf{type} \ni [p/\varphi]M_i}\ (\ni\text{-SINGLETON})$$

$$\frac{\Gamma \vdash p.\mathbf{type} \ni \mathsf{type}\ A = T \qquad \Gamma \vdash T \prec_\varphi \overline{M}}{\Gamma \vdash p.A \prec_\varphi \overline{M}}\ (\prec\text{-TYPE})$$

$$\frac{\Gamma \vdash T \prec_\varphi \overline{M} \qquad \varphi \notin \mathrm{fn}(M_i)}{\Gamma \vdash T \ni M_i}\ (\ni\text{-OTHER})$$

$$\frac{\Gamma \vdash p.\mathbf{type} \ni \mathsf{trait}\ A\ \mathsf{extends}\ S \qquad \Gamma \vdash S \prec_\varphi \overline{N}}{\Gamma \vdash p.A \prec_\varphi \overline{N}}\ (\prec\text{-CLASS})$$

Subtyping is reflexive, transitive, and obeys:

$$\frac{\Gamma \vdash p : T}{\Gamma \vdash p.\mathbf{type} <: T} \quad (\textsc{singleton-}<:)$$

$$\frac{\Gamma \vdash p : q.\mathbf{type}}{\Gamma \vdash q.\mathbf{type} <: p.\mathbf{type}} \quad (<:\textsc{-singleton})$$

$$\frac{\Gamma \vdash p.\mathbf{type} \ni \mathsf{type}\, A = S}{\Gamma \vdash p.A <: S} \quad (\textsc{type-}<:)$$

$$\frac{\Gamma \vdash p.\mathbf{type} \ni \mathsf{type}\, A = S}{\Gamma \vdash S <: p.A} \quad (<:\textsc{-type})$$

$$\frac{\Gamma \vdash p.\mathbf{type} \ni \mathsf{trait}\, A\, \mathsf{extends}\, S}{\Gamma \vdash p.A <: S} \quad (\textsc{class-}<:)$$

$$\frac{}{\Gamma \vdash \overline{T}\,\{\varphi \mid \overline{M}\} <: T_i} \quad (\textsc{sig-}<:)$$

$$\frac{\forall i,\ \Gamma \vdash S <: T_i \qquad \Gamma \vdash S \prec_\varphi \overline{M} \qquad \Gamma, \varphi : \overline{T}\,\{\varphi \mid \overline{N}\} \vdash \overline{M} \ll \overline{N}}{\Gamma \vdash S <: \overline{T}\,\{\varphi \mid \overline{N}\}} \quad (<:\textsc{-sig})$$

See Paper for ...

- Judgements for member subtyping $\ll$ and well-formedness 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 Algorthmic 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 type-check 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,
- Beta for systematic nesting,
- ML, Haskell for many of the functional aspects.
- OCaml, OHaskell, PLT-Scheme, as other combinations of FP and OOP.
- Pizza, Multi-Java, 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 : Bracha (linear), Duggan, Hirschkowitz (mixin-modules), Schaerli et al. (traits), Flatt et al. (units, Jiazzi), Zenger (Keris).

Abstract type members : Even more powerful are *virtual classes* (Cook, Ernst, Ostermann)

Explicit self-types : Vuillon and Rémy (OCaml)

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 $X, Y \in \{\text{FP, OOP}\}$.
- More things remain to be discovered if we look at symmetric combinations.
- Scala is one attempt to do so.

  Try it out: scala.epfl.ch

Thanks to the (past and present) members of the Scala team:

Philippe Altherr, Vincent Cremet, Julian Dragos, Gilles Dubochet, Burak Emir, Sean McDermid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, Matthias Zenger.