Featherweight Scala

Week 13/14
Dec 11/18

Today

Previously: Featherweight Java

Today: Featherweight Scala

- Live research, unlike what you have seen so far.
- Focus today on *path-dependent types*
- Plan:
  1. Rationale
  2. Examples
  3. Formal syntax and type checking

Based on...

This presentation is based on the slides of the talk:

> Martin Odersky, François Garillot, Vincent Cremet, and Sergueï Lenglet. "A Core Calculus for Scala Type Checking." Mathematical Foundations of Computer Science (MFCS), 2006.

Component Software – State of the Art

In *principle*, software should be constructed from re-usable parts ("components").

In *practice*, software is still most often written "from scratch", more like a craft than an industry.

Programming languages share part of the blame.

Most existing languages offer only limited support for components.

This holds in particular for statically typed languages such as Java and C#.

How To Do Better?

Hypothesis 1: Languages for components need to be *scalable*; the same concepts should describe small as well as large parts.

Hypothesis 2: Scalability can be achieved by unifying and generalizing *functional* and *object-oriented* programming concepts.

Why Unify FP and OOP?

Both have complementary strengths for composition:

Functional programming: Makes it easy to build interesting things from simple parts, using

- higher-order functions,
- algebraic types and pattern matching,
- parametric polymorphism.

Object-oriented programming: Makes it easy to adapt and extend complex systems, using

- subtyping and inheritance,
- dynamic configurations,
- classes as partial abstractions.

An Experiment

To validate our hypotheses we have designed and implemented a concrete programming language, **Scala**.

An open-source distribution of Scala has been available since Jan 2004.

Currently: $\approx$ 1000 downloads per month.

Version 2 of the language has been released Jan 2006.

A language by itself proves little; its applicability can only be validated by practical, serious use.

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;
}
```

Example: Peano Numbers...

```
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;
}

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

Example: Peano Numbers...

```
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;
}

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

Interoperability

Scala is completely interoperable with Java (and with some qualifications also to C#).

A Scala component can:

- access all methods and fields of a Java component,
- create instances of Java classes,
- inherit from Java classes and implement Java interfaces,
- be itself instantiated and called from a Java component.

None of this requires glue code or special tools.

This makes it very easy to mix Scala and Java components in one application.

Components

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.

Functors...

Functors have shortcomings, however:

- 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 [Odersky et al., MFCS06].

Scala's concepts subsume SML modules.

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

Component Abstraction

There are two principal forms of abstraction in programming languages:

parameterization (functional)

abstract members (object-oriented)

Scala supports both styles of abstraction for types as well as values.

Both types and values can be parameters, and both can be abstract members.

(In fact, Scala works with the *functional/OO duality* in that parameterization can be expressed by abstract members).

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

You can also use `AbsCell` without knowing the specific cell type:

```
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.

Example: Symbol Tables

As an example, let's look at extensible components of real compilers.

- Compilers need to model symbols and types.
- Each aspect depends on the other.
- Both aspects require substantial pieces of code.

The first attempt of writing a Scala compiler in Scala defined two global objects, one for each aspect:

First Attempt: Global Data

```
object Symbols {
  class Symbol {
    def tpe: Types.Type
    ...
  }
  // static data for
  // symbols}
}
```

```
object Types {
  class Type {
    def sym: Symbols.Symbol
    ...
  }
  // static data
  // for types
}
```

Problems:

1. `Symbols` and `Types` contain hard references to each other. Hence, impossible to adapt one while keeping the other.
2. `Symbols` and `Types` contain static data—not *reentrant*.

Second Attempt: Nesting

Static data can be avoided by nesting the `Symbols` and `Types` objects in a common enclosing class:

```
class SymbolTable {
  object Symbols {
    class Symbol { def tpe: Types.Type; ... }
  }
  object Types {
    class Type {def sym: Symbols.Symbol; ... }
  }
}
```

This solves the re-entrancy problem.

But we gave up separate compilation, and did not solve the reuse problem. There are still hard references to types.

Third Attempt: A Component-Based Solution

Question: How can one express the required services of a component?

Answer: By abstracting over them!

Two forms of abstraction: *parameterization* and *abstract members*.

Only abstract members can express recursive dependencies, so we will use them.

```
trait Symbols {
  type Type
  class Symbol { def tpe: Type }
}
```

```
trait Types {
  type Symbol
  class Type { def sym: Symbol }
}
```

`Symbols` and `Types` are now classes that each abstract over the identity of the "other type". How can they be combined?

Modular Mixin Composition

Here's how:

```
class SymbolTable extends Symbols with Types
```

Instances of the `SymbolTable` class contain all members of `Symbols` as well as all members of `Types`.

Concrete definitions in either base class override abstract definitions in the other.

Modular mixin composition generalizes the single inheritance + interfaces concept of Java and C#.
It is similar to *traits* [Schaerli et al, ECOOP 2003], but is more flexible since base classes may contain state.

Fourth Attempt: Mixins + Self-Types

The last solution modeled required types by abstract types.

This is limiting, because one cannot instantiate or inherit an abstract type.

A more general approach also makes use of *self-types*:

```
class Symbols { self: Symbols with Types =>
  class Symbol { def tpe: Type }
}
class Types { self: Types with Symbols =>
  class Type { def sym: Symbol }
}
class SymbolTable extends Symbols with Types
```

Self-types declare all of a components requirements.

Self-Types

- In a class declaration

  ```
  class C { self: T => ... }
  ```

  `T` is called a *self-type* of class `C`.

- The name `self` is arbitrary, it can also be a wildcard `_`.

- If a self-type is given, it is taken as the type of `this` inside the class.

- Without an explicit type annotation, the self-type is taken to be the type of the class itself.

Self-Types and Safety

- The self-type of a class must be a subtype of the self-types of all its base classes.

- When instantiating a class in a `new` expression, it is checked that the self-type of the class is a supertype of the type of the object being created.

Benefits

1. The presented scheme is very *general* – any combination of static modules can be lifted to a assembly of components.
2. Components have *documented interfaces* for required as well as provided services.
3. Components can be *multiply instantiated*

   $\Rightarrow$ *Re-entrancy* is no problem.
4. Components can be flexibly *extended* and *adapted*.

Foundations

A language like Scala is complicated.

How can we convince ourselves that types are sound... and can be computed?!

Featherweight Scala is a small calculus capturing:

- abstract type members
- path types
- mixins
- plus the usual OO dynamic dispatch

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
			$\mathsf{trait}\, A\, \mathsf{extends}\, \overline{T}\, \{\varphi \Rightarrow \overline{M}\}$	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
			$\overline{T}\, \{\varphi \Rightarrow \overline{M}\}$	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 =>
  type Elem = Nat
  val hd: Nat = zero.succ().succ()
  val tl: this0.ListOfElem =  new Nil { type Elem = Nat }
}
```

Type Assignment

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

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

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

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

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

Expansion and Membership

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

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

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

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

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

Subtyping is reflexive, transitive, and obeys:

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

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

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

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

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

$$\frac{\forall i,\ \Gamma \vdash S <: T_i \qquad \Gamma \vdash S \prec_\varphi \overline{M} \qquad \Gamma, \varphi : \overline{T}\left\{\varphi \Rightarrow \overline{N}\right\} \vdash \overline{M} \ll \overline{N}}{\Gamma \vdash S <: \overline{T}\left\{\varphi \Rightarrow \overline{N}\right\}} \quad (<:\text{-SIG})$$

$$\frac{}{\Gamma \vdash \overline{T}\left\{\varphi \Rightarrow \overline{M}\right\} <: T_i} \quad (\text{SIG-}<:)$$

Left out ...

This is not the whole formalism. In the paper there is also:

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

Future Work

1. Soundness proof for operational semantics

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
   - Existential types
   - 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)

In the other direction, Scala has its own influence. C# 3.0 has comprehensions, closures.... and "=>" :-)

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.
- Be ready to modify the techniques you have seen in this class.
- Theory and practice feed each other....