Week 13: Featherweight Scala

Martin Odersky

EPFL

Two Worlds

Objects and modules have complementary strengths.

- Modules are good at *abstraction*.

  For instance: abstract types in SML signatures.
  (Object systems offer only crude visibility control through modifiers such as **private** or **protected**).
- Objects are good at *composition*.

  For instance: Aggregation, recursion, inheritance, components as first-class values.
  (Only the first is supported by standard module systems).

Composition seems to be more popular than abstraction.

That's why mainstream languages use objects instead of modules, even though it comes at a cost in the expressiveness of types.

Can We Combine Both Worlds?

Idea: Identify

$$\begin{array}{lcl} \textit{Object} & \hat{=} & \textit{Module} \\ \textit{Interface} & \hat{=} & \textit{Signature} \\ \textit{Class} & \hat{=} & \textit{Functor} \end{array}$$

But then: Objects and interfaces need to contain *type members*.

Furthermore, type members can be either *abstract* or *concrete*.

Should We Combine Both Worlds?

Yes! Benefits are:

1. Better abstraction constructs for components
   (e.g. ML's signatures instead of Java's interfaces)
2. *Family polymorphism* is a powerful method of type specialization by overriding.

Example: Consider a family of types that represent *graphs*.

- A graph is given by the type of its nodes and the type of its edges.
- Both types should be refinable later.
- For instance nodes might have labels, or edges might have weights.

Here's a root class for graphs (Scala syntax).

```
abstract class Graph {
    type node <: Node;
    type edge <: Edge;

    class Node {
        val edges : List[edge];
        def neighbors : List[node] =
            edges.map { e => if (this == e.pred) e.succ else e.pred }
    }

    class Edge {
        val pred : node;
        val succ : node;
    }
}
```

- Nodes and edges are "bare-bone" abstractions in this class.
- However, they refer to each other via two *abstract types* edge and node.

Refining Graphs

A first refinement adds labels to nodes.

```
abstract class LabelledGraph extends Graph {
    type node <: LabelledNode;

    class LabelledNode extends Node {
        val label : String;
    }
}
```

Edges stay as they are.

In LabelledGraph, if e is an Edge, then e.pred refers to a LabelledNode or a subtype thereof.

The inherited neighbors method also returns a subtype of LabelledNode, instead of Node.

Refining Graphs Further

A second refinement adds weights to edges.

```
abstract class WeightedGraph extends Graph {
    type edge <: WeightedEdge;

    class WeightedEdge extends Edge {
        val weight : Int;
    }
}
```

We can also combine both refinements as follows:

```
abstract class WeightedLabelledGraph
    extends LabelledGraph with WeightedGraph {}
```


A Catch

- Because all graph classes contain abstract members node and edge, one cannot create directly graph objects, as in **new** Graph.
- One needs to bind the abstract members first, as in:

```
class MyGraph extends WeightedLabelledGraph {
    type node = LabelledNode;
    type edge = WeightedEdge;
}
val g = new MyGraph;
```

- One can imagine taking the bound of an abstract type as a default implementation; then the restriction becomes unnecessary.
- Most of the above can also be done using parameterized types, but at a cost of quadratic increase in the size of type variable bounds ⇒ Bruce, Odersky, Wadler, ECOOP 98.

Precedents

Has all this been tried?

Yes!

- Programming languages from Aarhus: Beta, more recently gbeta, Rune.
- Even more recently from Lausanne: Scala.

But what are the type-theoretic foundations?

- Intuition (Igarashi & Pierce): A type member T of an object referenced by r has the *dependent type* $r.T$.
- But aren't dependent types rather "hairy"?
- Problems: How to find good typing rules, and how to prove that they are sound.
- Precedent: SML-style module systems, but they'd need to be upgraded with first-class modules, inheritance and recursion.

What's Next

- We develop a type-systematic foundation of objects with dependent types.
- Objects can have type members.
- Such members may be concrete or abstract.
- They are referenced with expressions $p.T$ where
  - p is a *path*, i.e. an (immutable) identifier followed by zero or more field selections.
  - T is the name of a type in the object referenced by p
- These are called *path-dependent types*.

Path-Dependent Types

Question 1: Given

```
class C { type T; val m : this.T }
val c : C
```

what is the type of c.m?

Answer: c.T.

Question 2: Given a function

```
def f(): C = ...
```

what is the type of f().m? (it can't be f().T !)

Answer: f().m is not typable.

Question 3: Given,

```
class C { type T; val m : this.T }
class D extends C { type T = String }
val d : D
```

what is the type of d.m?

Answer: d.T or String (they are the same).

Question 4: Given a function

```
def g(): D = ...
```

what is the type of g().m?

Answer: String.

A Theory

We have developed a formal theory based on these intuitions.

Roadmap:

1. Construct *FS*, a basic calculus of nominal classes and objects.
2. Construct a *type system* for the calculus.
3. Extend *FS* to *FSG*, adding type members of objects.
4. Still to show:
   - Is the type system *sound* wrt the operational semantics?
   - Is type checking decidable?

Terms in *FS*

There are five forms of terms t in *FS*.

1. An *object-* or *class-reference* x.
2. The special **null** reference, which refers to no object or class.
3. A *selection* $t.l$ of a field l in an object denoted by t.
4. An *object creation* **val** $x =$ **new** $p\ ;\ t$

   Here, a *path* p is a name x followed by zero or more field selections.
5. A *class creation* **class** x **extends** $C\ ;\ t$.

   Here, C denotes a *class signature* (see next slide).

Definitions in *FS*

- A *class signature* $C ::= \overline{p}\,\{x\,|\,\overline{D}\;\overline{d}\}$ consists of:
  - a list of superclasses $\overline{p}$,
  - a self-reference x which names the current object,
  - a list of member declarations $\overline{D}$ and definitions $\overline{d}$.
- A *member declaration* D is of the form **val** $l\!:\!T$.
- A *member definition* d is of the form **val** $l\!:\!T = t$.

Types in *FS*

There are three forms of types T in *FS*

1. An *instance type* p.**inst**, which contains all instances created from class p.
2. A *singleton type* p.**type**, which contains the object or class referenced by p.
3. A *class type* **class** C, which contains all class values with a signature equal to C.

Additionally, the **null** reference forms part of the values of every type.

FS Syntax Summary

x, y, z			Name
l			Label
v, w	$::=$	$x \mid \mathbf{null}$	Value
s, t, u	$::=$		Term
		v	reference
		$t.l$	selection
		$\mathbf{val}\ x = \mathbf{new}\ p\ ;\ t$	new object
		$\mathbf{class}\ x\ \mathbf{extends}\ C\ ;\ t$	class definition
T	$::=$		Type
		$p.\mathbf{inst}$	class instance
		$p.\mathbf{type}$	singleton
		$\mathbf{class}\ C$	class type
C	$::=$	$\overline{p}\,\{x \mid \overline{D}\ \overline{d}\}$	Class signature
D	$::=$	$\mathbf{val}\ l : T$	Member declaration
d	$::=$	$\mathbf{val}\ l : T = t$	Member definition
M, N	$::=$	$D \mid d$	Member
p, q	$::=$	$v \mid p.l$	Path

Differences between FS and Scala

1. Different lexical conventions: Two namespaces (for terms and types) in Scala, unified namespace in FS.
2. The "self" reference of a class is denoted by a programmer-defined name in FS. Scala uses the reserved word **this**.
3. The instance type $p.$**inst** is simply written p in Scala (possible because of disjoint namespaces).
4. Scala does not have class types **class** C (even though analogous structures are maintained internally by the Scala compiler).

Encodings

Even though FS is much smaller than Scala, the majority of Scala's constructs can be mapped to it by *encodings*.

The encodings are quite direct because the essence of Scala's language constructs and its evaluation are modeled faithfully in FS.

In particular, there are objects created from classes, and there are classes built from base-classes using symmetric mixin composition; object equality is by reference and evaluation is strict.

Here are some of the encodings we will use:

Abbreviated instance types. We allow a path p to be used as a type, and expand it to $p.$**inst**.

Implicit self references. We allow to leave out the explicit self reference of a class signature.

A missing self reference is replaced by the predefined name *this*.

A class signature $\overline{p}\{\overline{D}\ \overline{d}\}$ is hence expanded to $\overline{p}\,\{this \mid \overline{D}\ \overline{d}\}$.

Value definitions. We introduce value definitions in terms.

$$t \quad ::= \quad \ldots \mid \mathbf{val}\ x : T = t\ ;\ u$$

Any such value definition is expanded as follows.

$$\mathbf{class}\ y\ \mathbf{extends}\ \{\mathbf{val}\ out : T = t\}\ ;\ \mathbf{val}\ x\ = \mathbf{new}\ y\ ;\ [x.out/x]u$$

Here, y is a fresh class name and out is a predefined label. As usual, $[x.out/x]u$ denotes the substitution of the term $x.out$ for every free occurrence of the name x in term u.

Eliding types. The type in a value definition may be elided, if it can be computed from the type of its right-hand side. E.g. $\textbf{val}\ l = t$ expands to $\textbf{val}\ l{:}T = t$ where T is the type of t. Note that this works only if t does not refer to the label l.

Adding types. We introduce terms annotated with types.

$$t \quad ::= \quad \ldots \mid t : T$$

Any such typed term is expanded to $\textbf{val}\ x : T = t\ ;\ x$ where x is a fresh name.

Anonymous classes. We permit to merge a class definition with exactly one superclass and an instantiation of that class. The merged term is usually called an anonymous class.

$$t \quad ::= \quad \ldots \mid \textbf{val}\ x = \textbf{new}\ p\,\{x \mid \overline{d}\}\ ;\ t\ \ .$$

Any such anonymous class is expanded to a sequence of a class definition and an object creation as follows.

$$\begin{array}{l} \textbf{class}\ y\ \textbf{extends}\ p\,\{x \mid \overline{d}\}\ ; \\ \textbf{val}\ x\ = \textbf{new}\ y\ ; \\ t : p.\textbf{inst} \end{array}$$

Here, y is a fresh name. Note that we have to explicitly annotate the type $p.\textbf{inst}$ of the final term t. Without such an annotation, the expansion would not be type-correct as the local class name y would escape into the type of the whole expansion.

New objects as terms. We permit $\textbf{new}\ p$ and $\textbf{new}\ C$ as terms. Any occurrence of a term $\textbf{new}\ p$ other than as a right hand side of a value definition $\textbf{val}\ x = \textbf{new}\ p$ is expanded to $\textbf{val}\ x = \textbf{new}\ p\ ;\ x$ where x is a fresh name. Analogously, anonymous class terms $\textbf{new}\ C$ are expanded to $\textbf{val}\ x = \textbf{new}\ C\ ;\ x$.

Class member definitions. We permit class definitions to be themselves class members:

$$d \quad ::= \quad \ldots \mid \textbf{class}\ l\ \textbf{extends}\ C\ \ .$$

Any such class definition is expanded as follows to a value definition containing a class definition on its right hand side.

$$\textbf{val}\ l{:}\textbf{class}\ C = (\textbf{class}\ x\ \textbf{extends}\ C\ ;\ x)\ \ ,$$

where x is a fresh name.

Methods. We introduce method types, method definitions and declarations, as well as method calls:

$$\begin{array}{lcl} T & ::= & \ldots \mid (T_1, \ldots, T_n) \Rightarrow U \\ d & ::= & \ldots \mid \textbf{def}\ l(x_1 : T_1, \ldots, x_n : T_n) : U = t \\ D & ::= & \ldots \mid \textbf{def}\ l(x_1 : T_1, \ldots, x_n : T_n) : U \\ t & ::= & \ldots \mid s(t_1, \ldots, t_n) \end{array}$$

A method definition such as the one in the first line above is expanded to a class definition as follows.

$$\begin{array}{l} \textbf{class}\ l\ \textbf{extends}\ \{ \\ \quad \textbf{val}\ in_1{:}T_1 \\ \quad \ldots \\ \quad \textbf{val}\ in_n{:}T_n \\ \quad \textbf{val}\ out{:}U = t \\ \} \end{array}$$

A method type $(T_1, \ldots, T_n) \Rightarrow U$ is expanded to a class type as follows.

$$\begin{array}{l} \textbf{class}\ \{ \\ \quad \textbf{val}\ in_1 : T_1 \\ \quad \ldots \\ \quad \textbf{val}\ in_n : T_n \\ \quad \textbf{val}\ out : U = \textbf{null} \\ \} \end{array}$$

Here, the abstract value definitions of $in_1, \ldots, in_n$ denote method parameters, whereas the concrete value definition *out* denotes the method result.

The result value as given in the type is **null**.

Concrete values of the function types can replace this value with an arbitrary term.

A method declaration

$$\textbf{def}\ l(x_1 : T_1, \ldots, x_n : T_n) : U$$

is expanded as follows to an abstract value definition:

$$\textbf{val}\ l : (T_1, \ldots, T_n) \Rightarrow U$$

Finally, a method call $s(t_1, \ldots, t_n)$ is expanded as follows.

$$\begin{array}{l} \textbf{val}\ x = s\ ; \\ \textbf{val}\ y = \textbf{new}\ x\ \{\textbf{val}\ in_1 = t_1 \ldots \textbf{val}\ in_n = t_n\}\ ; \\ y.out \end{array}$$

Constructors

Class constructors are encoded similarly to methods.

A class definition

$$\textbf{class}\ z(y_1 : T_1, \ldots, y_n : T_n)\ \textbf{extends}\ \overline{p}\ \{x \mid \overline{D}\ \overline{d}\}$$

is expanded to

$$\begin{array}{l} \textbf{class}\ z\ \textbf{extends}\ \overline{p}\ \{x \mid \\ \quad \textbf{val}\ in_1 : T_1 \\ \quad \ldots \\ \quad \textbf{val}\ in_n : T_n \\ \quad \overline{D} \quad [x.in_1/y_1, \ldots, x.in_n/y_n]\,\overline{d} \\ \} \end{array}$$

Constructor Calls

A class constructor call with arguments such as in

$$\textbf{new}\ z(t_1, \ldots, t_n)$$

is expanded to an anonymous class:

$$\begin{array}{l} \textbf{val}\ y_1 = t_1 \\ \ldots \\ \textbf{val}\ y_n = t_n \\ \textbf{new}\ z\ \{\textbf{val}\ in_1 = y_1, \ldots, \textbf{val}\ in_n = y_n\} \end{array}$$

The value definitions for $y_1, \ldots, y_n$ are necessary to maintain the correct order of evaluation: i.e. class constructor arguments are evaluated before members of the constructed object.

Superclass constructor calls are treated analogously.

The Global Environment

A *program* in *FS* is simply a term.

Realistic programs use a number of predefined libarary classes.

These classes have to be added to the main program as auxiliary class definitions.

Because of recursive dependencies between library classes, we cannot prefix them one by one to the main program in a sequence of class definitions.

Instead, we group all library classes as members of a common class *global*.

Here's a program consisting of a main expression t and library classes $c_1, \ldots, c_n$ with signatures $C_1, \ldots, C_n$:

$$
\begin{array}{l}
\textbf{class } \mathit{global}\ \{\mathit{root} \mid \\
\quad \textbf{class } c_1 \textbf{ extends } C_1 \\
\quad \ldots \\
\quad \textbf{class } c_n \textbf{ extends } C_1 \\
\}\ ; \\
\textbf{val } \mathit{root} = \textbf{new } \mathit{global}\ ; \\
t
\end{array}
$$

References to a library class are then always selections in the global environment *root*, e.g. $\mathit{root}.c_i$.

Differences between *FS* and *FJ*

- *FS* is more general; it has
  - nested and local classes,
  - value as well as method definitions,
  - singleton types,
  - class types.
- On the other hand, *FJ* has type casts – these could be added to *FS* without problem.
- *FS* is more realistic, since it has
  - strict evaluation
  - **null** references
  - reference equality
- *FS* is compositional – no global class environment.
- *FS* relies to a greater degree on encodings.

Differences between *FS* and Theory of Objects

Compared to Cardelli and Abadi's theory of objects, there are several important differences:

- There are classes besides objects and classes are first class terms.
- Objects can have type members
- The reduction relation of the calculus is based on *name passing*
  - This is necessary to maintain well-formedness of path-dependent types under reduction.
  - If we could replace a name (say x) by an arbitrary expression (say f()), then the legal type x.T would become the illegal type f().T after reduction.

Operational Semantics of *FS*

The operational semantics of *FS* is given by a big-step evaluation relation, using judgments of the form

$\Delta : t \Downarrow \Delta' : v$	In store Δ, term t evaluates to value v, with a resulting store Δ'.

There are two auxiliary judgment forms:

$\Delta \Downarrow_x \Delta'$	Initialization of object x in store Δ yields store Δ'.
$\Delta : y \prec_x \overline{d}$	In store Δ, the class named y has definitions $\overline{d}$, assuming x names the self-reference of the class.

Stores

A *store* Δ is a set of store-bindings $x \mapsto \delta$, which associate names with store-items.

$$\begin{array}{lcll} \Delta & ::= & \overline{x \mapsto \delta} \ \mid \ \Omega & \text{Store} \\ \delta & ::= & p(\overline{d}) \ \mid \ C & \text{Store binding} \end{array}$$

A store-item δ is either

- an object $p(\overline{d})$ of class p with members $\overline{d}$, or
- a class with signature C.

A special store value Ω denotes an store resulting from a run-time error leading to a program abort.

Evaluation Rules

(Val)

$$\Delta : v \Downarrow \Delta : v$$

(EvalClass)

$$\frac{\forall_{i=1}^{n} \quad \Delta : p_i \Downarrow \Delta : z_i \qquad \Delta, x \mapsto \overline{z}_{1..n}\,\{y \mid \overline{M}\} : t \Downarrow \Delta' : v}{\Delta : (\textbf{class}\ x\ \textbf{extends}\ \overline{p}_{1..n}\,\{y \mid \overline{M}\}\ ;\ t) \Downarrow \Delta' : v}$$

(EvalSel)

$$\frac{\Delta : t \Downarrow \Delta' : x \qquad l \notin \mathcal{L}(\overline{d'}) \qquad x \mapsto z(\overline{d}\ (\textbf{val}\ l{:}T = v)\ \overline{d'}) \in \Delta'}{\Delta : t.l \Downarrow \Delta' : v}$$

(EvalNew)

$$\frac{\Delta : p \Downarrow \Delta : z \qquad \Delta : z \prec_x \overline{d} \qquad \Delta, x \mapsto z(\overline{d}) \Downarrow_x \Delta' \qquad \Delta' : t \Downarrow \Delta'' : v}{\Delta : (\textbf{val}\ x = \textbf{new}\ p\ ;\ t) \Downarrow \Delta'' : v}$$

Here, $\mathcal{L}$ yields the label of a definition. I.e.

$$\mathcal{L}(\textbf{val}\ l{:}T) \ = \ l \qquad \mathcal{L}(\textbf{val}\ l{:}T = t) \ = \ l$$

Auxiliary Evaluation Rules

(Stable-Δ)

$$\frac{x \mapsto z(\overline{e}) \in \Delta}{\Delta \Downarrow_x \Delta}$$

(Unfold)

$$\frac{y \mapsto \overline{z}_{1...n}\,\{x \mid \overline{D}\ \overline{d}\} \in \Delta \qquad \forall_{i=1}^{n} \quad \Delta : z_i \prec_x \overline{d_i}}{\Delta : y \prec_x \overline{d_1} \ldots \overline{d_n}\ \overline{d}}$$

(Eval-Δ)

$$\frac{x \mapsto z(\overline{e}\ (\textbf{val}\ l{:}T = t)\ \overline{d}) \in \Delta \qquad \Delta : t \Downarrow \Delta' : v \qquad \Delta' \uplus (x \mapsto z(\overline{e}\ (\textbf{val}\ l : T = v)\ \overline{d})) \Downarrow_x \Delta''}{\Delta \Downarrow_x \Delta''}$$

Exceptional Evaluation Rules

The given evaluation rules are not yet sufficient since they do not talk about exceptional and error situations, like:

1. Selecting a member that has not yet been evaluated,
2. dereferencing **null**.
3. continuing in some error state

1. Unevaluated Members

During object initialization, one might select as-yet-unevaluated members.

Example:

class C { **val** x = y; **val** y = 1 }

or:

class C { **val** x = y; **val** y = x }

We handle this by returning **null** as a result of the selection:

(EvalSel')

$$\frac{\Delta : t \Downarrow \Delta' : x \qquad l \notin \mathcal{L}(\overline{d'}) \qquad u \notin \text{Value} \qquad x \mapsto z(\overline{d}\ (\mathbf{val}\ l : T = u)\ \overline{d'}) \in \Delta'}{\Delta : t.l \Downarrow \Delta' : \mathbf{null}}$$

2. Dereferencing Null

What is the result of a selection **null**.f?

In real life: throw a NullPointerException.

We approximate this by a result **null** and a store Ω, which signals "abort":

(SelNull)

$$\frac{\Delta : t \Downarrow \Delta' : \mathbf{null}}{\Delta : t.l \Downarrow \Omega : \mathbf{null}}$$

3. Propagating Error

We still need to define what happens when a store becomes Ω.

Under Ω, the program should abort with a **null** result.

This is expressed by the following rules:

(New-Null)

$$\frac{\Delta : p \Downarrow \Delta' : \mathbf{null}}{\Delta : (\mathbf{val}\ x = \mathbf{new}\ p\ ;\ t) \Downarrow \Omega : \mathbf{null}}$$

(New-Ω)

$$\frac{\Delta : p \Downarrow \Delta : z \qquad \Delta : z \prec_x \overline{d} \qquad \Delta, x \mapsto z(\overline{d}) \Downarrow_x \Omega}{\Delta : (\mathbf{val}\ x = \mathbf{new}\ p\ ;\ t) \Downarrow \Omega : \mathbf{null}}$$

(Cls-Null)

$$\frac{\exists_{i=1}^{n} \quad \Delta : p_i \Downarrow \Delta' : \mathbf{null}}{\Delta : (\mathbf{class}\ x\ \mathbf{extends}\ \overline{p}_{1..n}\ \{y \mid \overline{M}\}\ ;\ t) \Downarrow \Omega : \mathbf{null}}$$

(Eval-Ω)

$$\frac{x \mapsto z(\overline{e}\ (\mathbf{val}\ l : T = t)\ \overline{d}) \in \Delta \qquad \Delta : t \Downarrow \Omega : v}{\Delta \Downarrow_x \Omega}$$

Type System

The type system for *FS* specifies validity of the following main judgments:

$\Gamma \vdash t : T$	In environment Γ term t has type T.
$\Gamma \vdash T \ni M$	In environment Γ type T has a member M.
$\Gamma \vdash T <: T'$	In environment Γ type T conforms to type T'.
$\Gamma \vdash T = T'$	In environment Γ types T and T' are equal.
$\Gamma \vdash T\ \mathit{wf}$	In environment Γ type T is well formed (this is an abbreviation for $\Gamma \vdash T = T$).

The rules make use of a type environments Γ, which is a sequence of bindings of names to types:

$$\Gamma \quad ::= \quad \overline{x : T}$$

Type Assignment $\boxed{\Gamma \vdash t : T}$

(Var)

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

(Null)

$$\Gamma \vdash \mathbf{null} : \mathbf{null}.\mathbf{type}$$

(Path)

$$\frac{\Gamma \vdash p : T}{\Gamma \vdash p : p.\mathbf{type}}$$

(Sel)

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \ni (\mathbf{val}\ l : U)}{\Gamma \vdash t.l : U}$$

(Class)

$$\frac{x \notin fn(\Gamma, T) \qquad \Gamma \vdash C\ \mathit{wf} \qquad \Gamma, x : \mathbf{class}\ C \vdash t : T}{\Gamma \vdash (\mathbf{class}\ x\ \mathbf{extends}\ C\ ;\ t) : T}$$

(New)

$$\frac{x \notin fn(\Gamma, T) \qquad \Gamma \vdash p.\mathbf{inst} \mathrel{:\prec_x} \overline{d} \qquad \Gamma, x : p.\mathbf{inst} \vdash t : T}{\Gamma \vdash (\mathbf{val}\ x = \mathbf{new}\ p\ ;\ t) : T}$$

Membership $\boxed{\Gamma \vdash T \ni M}$

Two rules, depending whether T is a singleton type or not:

(Single-$\ni$)

$$\frac{x \notin fn(\Gamma) \qquad \Gamma \vdash p.\mathbf{type} \mathrel{:\prec_x} \overline{M_1}\ M\ \overline{M_2}}{\Gamma \vdash p.\mathbf{type} \ni [p/x]\mathcal{S}(M)}$$

(Other-$\ni$)

$$\frac{x \notin fn(\Gamma, M) \qquad \Gamma, x : T \vdash x.\mathbf{type} \ni M}{\Gamma \vdash T \ni M}$$

Here, $\mathcal{S}$ yields the declaration part (signature) of a definition. I.e.

$$\mathcal{S}(\mathbf{val}\ l : T) \;=\; \mathbf{val}\ l : T \qquad \mathcal{S}(\mathbf{val}\ l : T = t) \;=\; \mathbf{val}\ l : T$$

Membership rules use an auxiliary *"expansion"* judgement:

$\Gamma \vdash T \mathrel{:\prec_x} \overline{M}$	In environment Γ, type T has members $\overline{M}$, assuming that x is the self reference of T.

Expansion $\boxed{\Gamma \vdash T \mathrel{:\prec_x} \overline{M}}$ $\boxed{\Gamma \vdash C \mathrel{:\prec_x} \overline{M}}$

(Single-$:\prec$)

$$\frac{\Gamma \vdash p : T \qquad \Gamma \vdash T \mathrel{:\prec_x} \overline{M}}{\Gamma \vdash p.\mathbf{type} \mathrel{:\prec_x} \overline{M}}$$

(Inst-$:\prec$)

$$\frac{\Gamma \vdash p : \mathbf{class}\ C \qquad \Gamma \vdash C \prec_x \overline{M}}{\Gamma \vdash p.\mathbf{inst} \mathrel{:\prec_x} \overline{M}}$$

(ClsSig-$\prec$)

$$\frac{\forall_{i=1}^{n} \quad \Gamma \vdash p_i.\mathbf{inst} \mathrel{:\prec_x} \overline{D_i}\ \overline{d_i}}{\Gamma \vdash \overline{p}_{1\ldots n}\ \{x \mid \overline{D}\ \overline{d}\} \prec_x (\uplus_{i=1}^{n} \overline{D}_i) \uplus (\uplus_{i=1}^{n} \overline{d}_i) \uplus \overline{D}\ \overline{d}}$$

Rule (ClsSig-$\prec$) implements the membership rules of Scala. The following rules apply in the order they are given.

- Defined or declared members override inherited members.
- Concrete members override abstract members.
- Members inherited from later base classes override members inherited from earlier ones[a].

[a] in fact, Scala only prescribes that members from mixin classes override members inherited from the superclass.

Conformance

(Refl-<:)

$$\frac{\Gamma \vdash T = U}{\Gamma \vdash T <: U}$$

(Trans-<:)

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U}$$

(Single-<:)

$$\frac{\Gamma \vdash p : T}{\Gamma \vdash p.\mathbf{type} <: T}$$

(Null-<:)

$$\frac{\Gamma \vdash T \ \mathit{wf}}{\Gamma \vdash \mathbf{null.type} <: T}$$

(Inst-<:)

$$\frac{\Gamma \vdash p : \mathbf{class}\ \overline{q_1}\, q\, \overline{q_2}\, \{x \mid \overline{M}\}}{\Gamma \vdash p.\mathbf{inst} <: q.\mathbf{inst}}$$

(Defs-<:)

$$\frac{\forall_{i=1}^{k}\ \exists_{j=1}^{n}\quad \mathcal{S}(M_j) <: \mathcal{S}(N_i)}{\Gamma \vdash \overline{M}_{1..n} <: \overline{N}_{1..k}}$$

(Vdcl-<:)

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash (\mathbf{val}\ L : T) <: (\mathbf{val}\ L : U)}$$

Equality

The equality theory of types in *FS* is (mostly) straightforward. Most rules simply define a structural equality relation on terms.

There are two rules worth noting:

(Path-=)

$$\frac{\Gamma \vdash p : q.\mathbf{type}}{\Gamma \vdash p = q}$$

(Vdef-=)

$$\frac{\Gamma \vdash T = T' \qquad \Gamma \vdash t : U \qquad \Gamma \vdash t' : U' \qquad \Gamma \vdash U <: T \qquad \Gamma \vdash U' <: T'}{\Gamma \vdash (\mathbf{val}\ l : T = t) = (\mathbf{val}\ l : T' = t')}$$

Rule (Path-=) says that if p's declared type is q.**type**, then p and q are equal.

Rule (Vdef-=) says that two value definitions are equal if their names and declared types are equal; right hand sides don't matter.

Well-formedness

The well-formedness judgment $\Gamma \vdash X\ \mathit{wf}$ decides whether entity X is well-formed.

The most complicated rule is the well-formedness rule for class-signatures:

(ClsSig-=)

$$\frac{\begin{array}{c} x, y \notin \mathit{fn}(\Gamma) \qquad z \notin \mathit{fn}(\Gamma, x, y) \\ C \stackrel{\text{def}}{\equiv} \overline{p}\, \{x \mid \overline{M}\} \qquad \Gamma' \stackrel{\text{def}}{\equiv} \Gamma, z : \mathbf{class}\ C, x : z.\mathbf{inst} \\ \Gamma \vdash \overline{p}\ \mathit{wf} \qquad \Gamma \vdash C \prec_x \overline{N} \qquad \Gamma' \vdash \overline{N}\ \mathit{wf} \\ \forall_{i=1}^{n}\ \ \Gamma \vdash p_i.\mathbf{inst} :\prec_x \overline{N_i} \qquad \Gamma' \vdash \overline{N} <: \overline{N_i} \\ \forall_{i,j=1}^{k}\ \ i \neq j \ \Rightarrow\ \mathcal{L}(M_i) \neq \mathcal{L}(M_j) \end{array}}{\Gamma \vdash \overline{p}_{1..n}\, \{x \mid \overline{M}_{1..k}\} \mathit{wf}}$$

The Generic Calculus

So far, we cannot yet express (universal) polymorphism or abstract types.

To do this, we introduce *type members* in objects.

Syntax extension:

L			Type label
T	$::=$	$\ldots \mid T\#L$	type selection
D	$::=$	$\ldots \mid$ **type** $L <: T$	type declaration
d	$::=$	$\ldots \mid$ **type** $L = T$	type definition

The path-dependent type $p.T$ is then syntactic sugar for p.**type**$\#T$.

Type parameters can be encoded as abstract type members, similar to the way value parameters are encoded as abstract value members.

Evaluation

Evaluation rules are completely unchanged.

This means: type members are static; they do not form part in evaluation.

Expansion and Conformance

There's one new rule for member expansion, and there are three new rules for conformance:

(Tsel-:≺)

$$\frac{\Gamma \vdash T \ni (\mathbf{type}\ L \leq: U) \qquad \Gamma \vdash U :\prec_x \overline{M}}{\Gamma \vdash T\#L :\prec_x \overline{M}}$$

(Tsel-<:)

$$\frac{\Gamma \vdash T \ni (\mathbf{type}\ L <: U)}{\Gamma \vdash T\#L <: U}$$

(Tdcl-<:)

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash (\mathbf{type}\ L \leq: T) <: (\mathbf{type}\ L <: U)}$$

(Tdef-<:)

$$\frac{\Gamma \vdash T = U}{\Gamma \vdash (\mathbf{type}\ L = T) <: (\mathbf{type}\ L = U)}$$

Meta-Theory

Two results as yet to be shown:

Conjecture 1: Type soundness: If $\Gamma \vdash t : T$ and $\Gamma \vdash \Delta$ and $\Delta : t \Downarrow \Delta' : v$ then

$$\begin{array}{ll} (1) & \Gamma \vdash v : T \\ (2) & \exists \Gamma'.\ \Gamma, \Gamma' \vdash \Delta' \end{array}$$

Conjecture 2: Decidability: There is an algorithm to decide whether $\Gamma \vdash t : T$

Summary

- We have introduced Featherweight Scala, a simple calculus that expressed the essence of Scala.
- The calculus consists of a operational semantics and typing rules.
- Makes use of Duality: Functional constructions can be encoded in object-oriented.