A Nominal Theory of Objects with Dependent Types

Martin Odersky, Vincent Cremet, Christine Röckl, Matthias Zenger

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 currently more popular than abstraction.

That's why most popular languages are based on object systems, even though it comes at a cost in the expressiveness of types.

Can We Combine Both Worlds?

Idea: Identify

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

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. SML'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).

```
trait 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.

```
trait 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.

```
trait WeightedGraph extends Graph {
  type edge <: WeightedEdge;

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

We can also combine both refinements as follows:

```
trait 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 $\Rightarrow$ 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 to deal with first-class modules, inheritance and recursion.

Our Contribution

- 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

$$\textit{\textbf{class}}\ C\ \{\ \textit{\textbf{type}}\ T;\ \textit{\textbf{val}}\ m : \textit{\textbf{this}}.T\ \}$$
$$\textit{\textbf{val}}\ c : C$$

what is the type of $c.m$?

Answer: $c.T$.

Question 2: Given a function

$$\textit{\textbf{def}}\ f(\,): C = \ldots$$

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

Answer: $f(\,).m$ is not typable.

Question 3: Given,

$$\textbf{\textit{class}}\ C\ \{\ \textbf{\textit{type}}\ T;\ \textbf{\textit{val}}\ m : \textbf{\textit{this}}.T\ \}$$
$$\textbf{\textit{class}}\ D\ \textbf{\textit{extends}}\ C\ \{\ \textbf{\textit{type}}\ T = String\ \}$$
$$\textbf{\textit{val}}\ d : D$$

what is the type of $d.m$?

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

Question 4: Given a function

$$\textbf{\textit{def}}\ g(\,): D = \ldots$$

what is the type of $g(\,).m$?

Answer: $String$.

A Theory

We have developed a formal theory based on these intuitions.

Roadmap:

1. Construct νObj, a calculus of classes and objects with type members.
2. Construct a *type system* for the calculus.
3. Show that the type system is *sound* wrt the operational semantics.

νObj Terms

x, y, z	Name	roughly corresponds in Scala to:
l, m, n	Term label	
s, t, u ::=	Term	
x	Variable	
$t.l$	Selection	
$\nu x \leftarrow t \; ; \; u$	New object	$\cong$ x = ***new*** t ; u
$[x : S \mid \overline{d}]$	Class template	$\cong$ the value defined by ***class*** S { $\overline{d}$ }
$t \;\&_S\; u$	Composition	$\cong$ tvdb ***class*** S ***extends*** t ***with*** u
d ::=	Definition	
$l = t$	Term definition	
$L \preceq T$	Type definition	
p ::=	Path	
$x \mid p.l$		
v ::=	Value	
$x \mid [x : S \mid \overline{d}]$		

What's Missing

- Functions and function abstractions can be encoded using classes.
- Parameterized types are encoded as types with abstract type members:

$$\textbf{\textit{type}}\ S = \{\ \textbf{\textit{type}}\ T;\ \ldots\ \} \qquad \text{encodes} \qquad \textbf{\textit{type}}\ S[T] = \{\ \ldots\ \}.$$

- Polymorphic functions are encoded as classes with abstract type members.
- In this way the whole of $F_{<:}$ can be encoded in νObj.

What's Different

Compared with 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*, (more accurately: paths are rewritten to other paths).
  - 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 νObj

Reduction

$$\text{(select)} \qquad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \;;\; e\langle x.l \rangle \quad \rightarrow \quad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \;;\; e\langle v \rangle$$

Operational Semantics of νObj

Reduction

$$\text{(select)} \qquad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \ ; \ e\langle x.l \rangle \quad \to \quad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \ ; \ e\langle v \rangle$$

$$\text{(combine)} \qquad [x : S_1 \mid \overline{d}_1] \ \&_S \ [x : S_2 \mid \overline{d}_2] \quad \to \quad [x : S \mid \overline{d}_1 \uplus \overline{d}_2]$$

Operational Semantics of νObj

Reduction

$$\text{(select)} \qquad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \; ; \; e\langle x.l \rangle \quad \rightarrow \quad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \; ; \; e\langle v \rangle$$

$$\text{(combine)} \qquad [x : S_1 \mid \overline{d}_1] \;\&_S\; [x : S_2 \mid \overline{d}_2] \quad \rightarrow \quad [x : S \mid \overline{d}_1 \uplus \overline{d}_2]$$

Structural Equivalence

α-renaming of bound variables x, plus

$$\text{(extrude)} \qquad e\langle \nu x \leftarrow t \; ; \; u \rangle \quad \equiv \quad \nu x \leftarrow t \; ; \; e\langle u \rangle$$

Operational Semantics of νObj

Reduction

$$\text{(select)} \qquad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \; ; \; e\langle x.l \rangle \quad \rightarrow \quad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \; ; \; e\langle v \rangle$$

$$\text{(combine)} \qquad [x : S_1 \mid \overline{d}_1] \;\&_S\; [x : S_2 \mid \overline{d}_2] \quad \rightarrow \quad [x : S \mid \overline{d}_1 \uplus \overline{d}_2]$$

Structural Equivalence α-renaming of bound variables x, plus

$$\text{(extrude)} \qquad e\langle \nu x \leftarrow t \; ; \; u \rangle \quad \equiv \quad \nu x \leftarrow t \; ; \; e\langle u \rangle$$

where evaluation context

$$e \quad ::= \quad \langle\rangle \mid e.l \mid e \;\&_S\; t \mid t \;\&_S\; e \mid \nu x \leftarrow t \; ; \; e \mid \nu x \leftarrow e \; ; \; t \mid \nu x \leftarrow [x : S \mid \overline{d}, l = e] \; ; \; t$$

Theorem: $\rightarrow$ is confluent:

If $t \twoheadrightarrow t_1$ and $t \twoheadrightarrow t_2$ then there exists a term t' such that $t_1 \twoheadrightarrow t'$ and $t_2 \twoheadrightarrow t'$.

Nominal Types

The type system should be able to express the nominal nature of classes and interfaces in object-oriented languages.

That is, two type or interface definitions with the same body should (be able to) yield different types.

Reasons:

- That's how most languages in use work.
- Nominal types help avoid accidental type identifications.
- Nominal types make it feasible to type-check recursive dependent types, which can be non-regular.

Nominal Type Bindings

We introduce three type bindings, one of which is nominal.

$L = T$	The type label L is an **alias** for the type T. That is, the two are interchangeable.
$L \prec T$	The type label L represents a **new** type which expands (or: unfolds) to type T.
$L <: T$	The type label L represents an **abstract** type which is bounded by type T.

The right hand side of a $\prec$ or $<:$ binding can be recursive.

By contrast, recursive aliases are disallowed.

Example: Here's a simple type definition for lists of integers.

$$List \prec \{\ isEmpty\colon Boolean,\ head\colon Int,\ tail\colon List\ \}$$

$List$ is the name of a nominal type.

Two aspects of $(\prec)$:

- $List$ is a subtype of the *record type* $\{\ isEmpty\colon Boolean,\ head\colon Int,\ tail\colon List\ \}$.
- Objects of type $List$ can be created from classes that define fields $isEmpty$, $head$ and $tail$ with the given types.

Record Types

Generally, a **record type** has the form $\{x \mid \overline{D}\}$, where

- $\overline{D}$ is a list of value declarations $l : T$ or type declarations $L \preceq: T$.
- x is a name for the object itself (i.e. an α-renamable version of ***this***).
- It can be omitted if not referenced.

References of one declaration to another always go via self, i.e.

$$\{ \; x \mid L = String; \; m : x.L \; \}$$

Question: How does one create a subtype of nominal type?

Example: Let's create a type for lists with a length operation.

First attempt:

$$\begin{aligned} \mathit{ListWithLen} \prec \{\ & \mathit{isEmpty}\colon \mathit{Boolean}, \\ & \mathit{head}\colon \mathit{Int}, \\ & \mathit{tail}\colon \mathit{ListWithLen}, \\ & \mathit{length}\colon \mathit{Int}\ \} \end{aligned}$$

In this case, $\mathit{ListWithLen}$ is a subtype of List's expansion, $\{\ \mathit{isEmpty}\colon \mathit{Boolean},\ \mathit{head}\colon \mathit{Int},\ \mathit{tail}\colon \mathit{List}\ \}$.

But it is not a subtype of $\mathit{ListWithLen}$ itself.

Compound Types

A subtype of a nominal type takes the form of a **compound type**.

Example:

$$ListWithLen \prec List \;\&\; \{\, tail : ListWithLen,\ length : Int \,\}$$

is a subtype of $List$ as well as $\{\, tail : ListWithLen,\ length : Int \,\}$.

It has four fields:

> $isEmpty : Boolean$ and $head : Int$, which come from $List$,
> $tail : ListWithLen$,
> $length : Int$.

The compound type operator $\&$ behaves like type intersection wrt subtyping, but its formation rule is more restrictive:

If in $T \,\&\, U$ a label is bound in both T and U, then the binding in U must be more specific than the binding in T.

νObj Terms and Types

x, y, z	Name		
l, m, n	Term label	L, M, N	Type label
$s, t, u \ ::=$	Term	$S, T, U \ ::=$	Type
$\quad x$	$\quad$ Variable	$\quad p.\mathbf{type}$	$\quad$ Singleton
$\quad t.l$	$\quad$ Selection	$\quad T \bullet L$	$\quad$ Type selection
$\quad \nu x \leftarrow t \ ; \ u$	$\quad$ New object	$\quad \{x \mid \overline{D}\}$	$\quad$ Record type $(=:: \ R)$
$\quad [x : S \mid \overline{d}]$	$\quad$ Class template	$\quad [x : S \mid \overline{D}]$	$\quad$ Class type
$\quad t \ \&_S \ u$	$\quad$ Composition	$\quad T \ \& \ U$	$\quad$ Compound type
$d \quad ::=$	Definition	$D \quad ::=$	Declaration
$\quad l = t$	$\quad$ Term definition	$\quad l : T$	$\quad$ Term declaration
$\quad L \preceq T$	$\quad$ Type definition	$\quad L \preceq: T$	$\quad$ Type declaration
$p \quad ::=$	Path	$\preceq: \quad ::=$	Type binder
$\quad x \mid p.l$		$\quad =$	$\quad$ Type alias
$v \quad ::=$	Value	$\quad \prec$	$\quad$ New type
$\quad x \mid [x : S \mid \overline{d}]$		$\quad <:$	$\quad$ Abstract type

Typing Judgments

$\Gamma \vdash t : T$	Term t has type T in environment Γ.

(An environment Γ is a finite set of bindings $x : T$, where the x are pairwise different.)

Auxiliary Judgments

$\Gamma \vdash T\ wf$	Type T is well-formed.
$\Gamma \vdash T \ni D$	Type T contains declaration D.
$\Gamma \vdash T \prec U$	Type T expands to type U.
$\Gamma \vdash T \leq U$	Type T is a subtype of type U.

Type Assignment

(Var)

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

(Sel)

$$\frac{\Gamma \ \vdash \ t : T, \quad T \ni (l : U)}{\Gamma \ \vdash \ t.l : U}$$

(VarPath)

$$\frac{\Gamma \ \vdash \ x : R}{\Gamma \ \vdash \ x : x.\mathbf{type}}$$

(SelPath)

$$\frac{\Gamma \ \vdash \ t : p.\mathbf{type}, \quad t.l : R}{\Gamma \ \vdash \ t.l : p.l.\mathbf{type}}$$

(Sub)

$$\frac{\Gamma \ \vdash \ t : T, \quad T \leq U}{\Gamma \ \vdash \ t : U}$$

(New)

$$\frac{\begin{array}{c}\Gamma \ \vdash \ t : [x : S \mid \overline{D}], \quad S \prec \{x \mid \overline{D}\} \\ \Gamma, x : S \ \vdash \ u : U \qquad x \notin \mathrm{fn}(U)\end{array}}{\Gamma \ \vdash \ (\nu x \leftarrow t \ ; \ u) : U}$$

(Class)

$$\frac{\begin{array}{c}\Gamma \ \vdash \ S \ \mathit{wf} \qquad \Gamma, x : S \ \vdash \ \overline{D} \ \mathrm{wf}, \quad t_i : T_i \\ t_i \ \mathit{contractive\ in}\ x \qquad (i \in 1..n)\end{array}}{\Gamma \ \vdash \ [x : S \mid \overline{D}, l_i = t_i{}^{i \in 1..n}] : [x : S \mid \overline{D}, l_i : T_i{}^{i \in 1..n}]}$$

(&)

$$\frac{\begin{array}{c}\Gamma \ \vdash \ t_i : [x : S_i \mid \overline{D}_i] \\ \Gamma \ \vdash \ S \ \mathrm{wf}, \quad S \leq S_i \qquad (i = 1, 2)\end{array}}{\Gamma \ \vdash \ t_1 \ \&_S \ t_2 : [x : S \mid \overline{D}_1 \uplus \overline{D}_2]}$$

The Essence of Path Dependent Types

- Judgment: $\Gamma \vdash T \ni D$ Type T contains declaration D.
- Two rules, depending whether T is a singleton type or not:

(Single-$\ni$) (Other-$\ni$)

$$\frac{\Gamma \vdash p.\mathbf{type} \leq \{x \mid \overline{D'}, D\}}{\Gamma \vdash p.\mathbf{type} \ni [p/x]D} \qquad \frac{\Gamma, x : T \vdash x.\mathbf{type} \ni D \qquad x \notin fn(\Gamma, D)}{\Gamma \vdash T \ni D}$$

- For T to contain a declaration, it must be a subtype of a record type.
- If T is a singleton type $p.\mathbf{type}$, then we replace the self-identity x in the record type by p.
- If T is not a singleton type, we invent a fresh variable $x : T$ and derive a judgment $\Gamma \vdash x.\mathbf{type} \ni D$.
- In this case, the resulting D is not allowed to refer to x.

Properties of νObj

Theorem: [Subject Reduction] If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.

Theorem: [Type Soundness] If $\vdash t : T$ then either $t\Uparrow$ or $t \twoheadrightarrow a$, for some answer a such that $\vdash a : T$.

Theorem: It is undecidable whether $\Gamma \vdash t : T$.

Proof by reduction to the problem in $F_{<:}$.

Summary

νObj is a nominal theory of objects with dependent types.

Nominal means two things:

- The operational semantics uses name passing instead of value passing.
- There is a type binder $\prec$, which creates nominal types.

The theory can express among others

- Nominal interface types, as in Java.
- Virtual types and family polymorphism.
- Generative SML structures and functors.
- A solution to the "expression problem".

More importantly, the theory can be used as a basis for existing and future languages that unify these elements in more flexible and precise constructs for composition and abstraction.

Notes

- $\uplus$ is concatenation with overwriting of common labels:

$$\overline{a} \uplus \overline{b} = \overline{a}|_{dom(\overline{a})\setminus dom(\overline{b})}, \overline{b}.$$

- Side conditions on reduction rules ensure that free variables are not captured.
- Reduction $\rightarrow$ is the smallest reflexive (wrt $\equiv$) and transitive relation that satisfies rules (select) and (mix) and that is closed under formation of evaluation contexts:

$$t \rightarrow u \qquad \text{implies} \qquad e\langle t\rangle \rightarrow e\langle u\rangle$$

Theorem: $\rightarrow$ is confluent:

If $t \twoheadrightarrow t_1$ and $t \twoheadrightarrow t_2$ then there exists a term t' such that $t_1 \twoheadrightarrow t'$ and $t_2 \twoheadrightarrow t'$.

Other Type Constructors

Besides record types and compound types, there are three more type constructors in νObj.

- A **class type** $[x : S \mid \overline{D}]$ which defines members $\overline{D}$ and which is used to create objects of type S.

  Members of S that are not in $\overline{D}$ are abstract; they need to be defined before an object of the class can be created.

- A **singleton type** $p.\mathbf{type}$ which represents the set consisting of just the object referenced by path p.

- A **type selection** $T{\bullet}L$ which references the type member named L in type T.

  The path dependent type $p.L$ is syntactic sugar for $p.\mathbf{type}{\bullet}L$.