A Nominal Theory of Objects with Dependent Types

Martin Odersky, Vincent Cremet, Christine Röckl, Matthias Zenger

EPFL

Can We Combine Both Worlds?

<table>
<thead>
<tr>
<th>Idea</th>
<th>Identify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Module</td>
</tr>
<tr>
<td>Interface</td>
<td>Signature</td>
</tr>
<tr>
<td>Class</td>
<td>Functor</td>
</tr>
</tbody>
</table>

But then: Objects and interfaces need to contain type members. Furthermore, type members can be either abstract or concrete.

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.

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 overloading.

Examples: 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.

Refining Graphs

A first refinement adds labels to nodes.

```scala
trait Graph {
  type node < Node;
  type edge < Edges;
  class Node {
    def edges: List[Edge];
    def neighbors: List[Node] = 
    edges.foreach { e => if (this == e.src) erase else push }
  }
  class Edge {
    def src: node;
    def dest: node;
  }
}
```

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

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

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

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

```scala
trait Graph {
  type node < Node;
  type edge < Edges;
  class Node {
    def edges: List[Edge];
    def neighbors: List[Node] = 
    edges.foreach { e => if (this == e.src) erase else push }
  }
  class Edge {
    def src: node;
    def dest: node;
  }
}
```
Refining Graphs Further

A second refinement adds weights to edges:

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

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

We can also combine both refinements as follows:

```scala
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:

```scala
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, Odellsø, Wadler, ECOOP 98.

Precedents

Has all this been tried?

Yes!

- Programming languages from Aarhus/Beta, more recently ghc, Java, ... Even more recently from Lucent’s Scala.

But what are the type-theoretic foundations?

- Intuition (Girard & 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-theoretic 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

```scala
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

```scala
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:

```scala
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

```scala
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 \( \nu \text{Obj} \), 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.

\( \nu \text{Obj} \) Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>Variable</td>
</tr>
<tr>
<td>( t )</td>
<td>Selection</td>
</tr>
<tr>
<td>( [x:S \mid \ell] )</td>
<td>New object</td>
</tr>
<tr>
<td>( \ell )</td>
<td>Class template</td>
</tr>
<tr>
<td>( \ell )</td>
<td>Composition</td>
</tr>
</tbody>
</table>

\( \lambda \) and \( \eta \) roughly corresponds to Scala \( \text{for} \) and \( \text{with} \).

What’s Missing

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

\[
\text{type } S = \{ \text{type } T \mid \ldots \} \quad \text{encodes} \quad \text{type } S[T] = \{ \ldots \}.
\]
- Polymorphic functions are encoded as classes with abstract type members.
- In this way the whole of \( F_{\lambda \eta} \) can be encoded in \( \nu \text{Obj} \).

What’s Different

Compared with Cardelli and Hudak’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 \( f(x) \).

Operational Semantics of \( \nu \text{Obj} \)

Reduction

\[
\begin{align*}
\text{(select)} & \quad \nu x \leftarrow [x:S \mid \ell, \ell = v] ; e(x, y) \quad \rightarrow \quad \nu x \leftarrow [x:S \mid \ell, \ell = v] ; e(y)
\end{align*}
\]
Operational Semantics of \(\nu\)Obj

Reduction

(select) \(\nu x \leftarrow [x:S | \mathcal{L} = v] ; e(x) \rightarrow \nu x \leftarrow [x:S | \mathcal{L} = v] ; e(v)\)

(combine) \([x:S | \mathcal{L}_1 \& S | x:S | \mathcal{L}_2] \rightarrow [x:S | \mathcal{L}_1 \& \mathcal{L}_2]\)

Structural Equivalence \(\alpha\)-renaming of variables, plus

(extract) \(e(x\leftarrow t ; u) \equiv x\leftarrow t ; e(u)\)

Theorem: \(\rightarrow\) is confluent:

If \(t \rightarrow t_1\) and \(t \rightarrow t_2\) then there exists a term \(t'\) such that \(t_1 \rightarrow t'\) and \(t_2 \rightarrow t'\).

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 \(T\).

\[ L \equiv T \] 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 \(<\) or \(<=\) binding can be recursive.

By contrast, recursive aliases are disallowed.

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.

Examples: Here’s a simple type definition for lists of integers.

\[ \text{List} \to \{ \text{isEmpty}, \text{head}, \text{Int}, \text{tail}, \text{List} \} \]

\(\text{List}\) is the name of a nominal type.

Two aspects of \(\to\):

- \(\text{List}\) is a subtype of the \textit{record type}
  \[ \{ \text{isEmpty}, \text{head}, \text{Int}, \text{tail}, \text{List} \} \]
- Objects of type \(\text{List}\) can be created from classes that define fields
  \(\text{isEmpty}, \text{head}\) and \(\text{tail}\) with the given types.
Record Types

- A record type has the form \( \{ x \mid D \} \), where
- \( D \) is a list of value declarations \( I : T \) or type declarations \( L =: T \).
- \( x \) is a name for the object itself (i.e., an \( n \)-th-amenable 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:

\[
\text{ListWithLen} = \{ \text{isEmpty} : \text{Boolean}, \text{head} : \text{Int}, \text{tail} : \text{ListWithLen}, \text{length} : \text{Int} \}
\]

In this case, \( \text{ListWithLen} \) is a subtype of \( \text{List} \)’s expansion, \( \{ \text{isEmpty} : \text{Boolean}, \text{head} : \text{Int}, \text{tail} : \text{List} \} \).

But it is not a subtype of \( \text{ListWithLen} \) itself.

Compound Types

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

Example:

\[
\text{ListWithLen} = \{ \text{tail} : \text{ListWithLen}, \text{length} : \text{Int} \}
\]

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

It has four fields:

- \( \text{isEmpty} : \text{Boolean} \) and \( \text{head} : \text{Int} \), which come from \( \text{List} \),
- \( \text{tail} : \text{ListWithLen} \),
- \( \text{length} : \text{Int} \).

The compound type operator \& behaves like a type intersection with 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 \( T \).

Typing Judgments

- \( \Gamma \vdash t : T \) Term \( t \) has type \( T \) in environment \( \Gamma \).

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

Auxiliary Judgments

- \( \Gamma \vdash T \text{ wf} \) Type \( T \) is well-formed.
- \( \Gamma \vdash T \supset D \) Type \( T \) contains declaration \( D \).
- \( \Gamma \vdash T \sim U \) Type \( T \) expands to type \( U \).
- \( \Gamma \vdash T \leq U \) Type \( T \) is a subtype of type \( U \).

Type Assignment

- (Var)

\[
\text{Var} \quad \Gamma \vdash z : B \\
\Gamma \vdash z : T \\
\Gamma, z : T \vdash t : U \\
\Gamma \vdash t : U
\]

(VarP)(\text{VarP})

\[
\text{VarP} \quad \Gamma \vdash z : B \\
\Gamma \vdash z : \text{type} \\
\Gamma \vdash z : T \\
\Gamma \vdash t : \text{type}
\]

(Sub)

\[
\text{Sub} \quad \Gamma \vdash t : T, T \equiv (u : U) \\
\Gamma \vdash t : U
\]

(SubP)(\text{SubP})

\[
\text{SubP} \quad \Gamma \vdash t : \text{type}, u : R \\
\Gamma \vdash t : \text{type}
\]

(New)

\[
\text{New} \quad \Gamma \vdash t : T, T \equiv (t : T) \\
\Gamma \vdash t : (p : T) \\
\Gamma \vdash t : (p : T) \vdash t : T
\]

(Class)

\[
\text{Class} \quad \Gamma \vdash S \text{ wf} \\
\Gamma \vdash t : T, T \equiv (t : T) \\
\Gamma \vdash t : T =: T' \text{\ in } \Gamma \\
\Gamma \vdash t : T =: T' \text{\ in } \Gamma \\
\Gamma \vdash t : T =: T' \text{\ in } \Gamma
\]

(k)

\[
\text{k} \quad \Gamma \vdash t : T, T \equiv (t : T) \\
\Gamma \vdash t : T \\
\Gamma \vdash t \vdash (t : T)
\]
The Essence of Path Dependent Types

• Judgment: \( \Gamma \vdash T \Rightarrow D \) Type \( T \) contains declaration \( D \).
• Two rules, depending whether \( T \) is a singleton type or not:

- \( \text{Single}_{\Rightarrow} \)
  \[
  \Gamma \vdash \text{p\text{type}} \subseteq \{ x \mid T, D \} \quad \Gamma, x : T \vdash \text{x\text{type}} \Rightarrow D \quad x \notin \mathcal{F}(\Gamma, D) \\
  \Gamma \vdash \text{p\text{type}} \Rightarrow [p/x]D \\
  \Gamma \vdash T \Rightarrow D
  \]

- \( \text{Other}_{\Rightarrow} \)
  \[
  \Gamma \vdash \text{p\text{type}} \subseteq \{ x \mid [T, D] \} \quad \Gamma, x : T \vdash \text{x\text{type}} \Rightarrow D \\
  \Gamma \vdash T \Rightarrow D
  \]

• For \( T \) to contain a declaration, it must be a subtype of a record type.
• If \( T \) is a singleton type \( \text{p\text{type}} \), 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 \( \Gamma \vdash x\text{\text{type}} \Rightarrow D \).
• In this case, the resulting \( D \) is not allowed to refer to \( x \).

Properties of \( \nu\text{Obj} \)

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 \( t \Downarrow \) or \( t \Rightarrow a \), for some answer \( a \) such that \( t \Downarrow a : T \).

Theorem: It is undecidable whether \( \vdash t : T \).
Proof by reduction to the problem in \( \text{F}_\omega \).

Summary

\( \nu\text{Obj} \) is a nominal theory of objects with dependent types.
Nominal terms two things:
• The operational semantics uses name passing instead of value passing.
• There is a type binder \(-\), 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 functions.
• A solution to the "expression problem".
More importantly, the theory can be used as a basis for existing and future languages that unify elements in more flexible and precise constructs for composition and abstraction.

Notes

• \( \text{if} \) is concatenation with overwriting of common labels:
  \[
  \pi_{\text{if}} \downarrow = \pi_{\text{if}}(\pi_{\text{if}}) \downarrow(\pi_{\text{if}}) \downarrow(\pi_{\text{if}}) \downarrow(\pi_{\text{if}}) \downarrow
  \]
• Side conditions on reduction rules ensure that free variables are not captured.
• Reduction \( \Rightarrow \) is the smallest reflective (\( \Rightarrow \)) and transitive relation that satisfies rules (select) and (mix) and that is closed under formation of evaluation contexts:
  \[
  t \Rightarrow u \quad \text{implies} \quad e(t) \Rightarrow e(u)
  \]
Theorem: \( \Rightarrow \) is confluent:
If \( t \Rightarrow t_1 \) and \( t \Rightarrow t_2 \) then there exists a term \( t' \) such that \( t_1 \Rightarrow t' \) and \( t_2 \Rightarrow t' \).

Other Type Constructors

Besides record types and compound types, there are three more type constructors in \( \nu\text{Obj} \).
• A class type \( [x : S | T] \) which defines members \( D \) and which is used to create objects of type \( S \).
  Members of \( S \) that are not in \( T \) are abstract; they need to be defined before an object of the class can be created.
• A singleton type \( \text{p\text{type}} \) which represents the set consisting of just the object referenced by path \( p \).
• A type selection \( T \cdot L \) which references the type member named \( L \) in type \( T \).
  The path dependent type \( p \cdot L \) is syntactic sugar for \( \text{p\text{type}} \cdot L \).