Types for Objects and Modules

Martin Odersky
École Polytechnique Fédérale de Lausanne

(joint work with Vincent Cremet, Christine Röckl, Matthias Zenger)

Dagstuhl, March 2003
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

\[
\begin{align*}
\text{Object} & \triangleq \text{Module} \\
\text{Interface} & \triangleq \text{Signature} \\
\text{Class} & \triangleq \text{Functor}
\end{align*}
\]

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).

```scala
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.

```scala
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.

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

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

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 WeightedLabelledGraph`.
- One needs to bind the abstract members first, as in:

```scala
val graph = new WeightedLabelledGraph {
  type node = LabelledNode;
  type edge = WeightedEdge;
}
...
new graph<Edge(...)>
```

- 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 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

```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,

```java
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

```java
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 $\nu$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.

See also:

FOOL 10, ECOOP 2003.
An Application

$vObj$ is one of very few type systems that can solve the expression problem.

This problem is about type-safe extensions of languages and compilers.

Say, you are given a language $L_1$ of arithmetic expressions and an interpreter that evaluates it.

Now, you might want to

- extend the language with a new operator, yielding $L_2$,
- add a new processor, such as a pretty-printer.

Two requirements:

Safety: Only type-safe combinations should be permitted, so applying the $L_1$-interpreter to a $L_2$ expression tree should give a static type error.

Re-use: Extensions should not change or duplicate existing code.
Non-Solutions

Any one of these extensions is easy, provided we have planned ahead for it.

- Adding a new language construct: Easy with standard object-oriented decomposition.
  Node types are classes, operations are methods.
- Adding a new language processor: Easy with functional decomposition.
  Node types form an algebraic data type, operations are pattern-matching functions.
  Or: Node types are classes, operations are encapsulated in a visitor.

But:

Much harder to provide extensibility in both directions.
Object-Oriented Decomposition

```java
class Lang {
    trait Exp { def eval: Int }
    class Num(n: int) extends Exp { def eval = n }
}

class Lang2 extends Lang {
    class Plus(l: Exp, r: Exp) extends Exp { def eval = l.eval + r.eval }
}

object Main {
    val l1 = new Lang;
    val e1: l1.Exp = new l1.Num(42);
    System.out.println("eval: " + e1.eval);

    val l2 = new Lang2;
    System.out.println("eval: " + e2.eval);
}
```
Visitor Decomposition

class Lang {
    trait Visitor[a] {  def caseNum(n: int): a }
    trait Exp {
        def visit[a](v: Visitor[a]): a;
    }
    class Num(n: int) extends Exp {
        def visit[a](v: Visitor[a]): a = v.caseNum(n);
    }
    class Eval extends Visitor[Int] {
        def caseNum(n: int) = n
    }
}

class Lang2 extends Lang {
    class Show extends Visitor[String] {
        def caseNum(n: int) = n.toString()
    }
}
object Main {
    val l1 = new Lang;
    val e1: l1.Exp = new l1.Num(42);
    System.out.println("eval: "+ e1.visit(new l1.Eval));

    val l2 = new Lang2;
    val e2: l2.Exp = new l2.Num(37);
    System.out.println("show: "+ e2.visit(new l2.Show));
}
Solution

We need to abstract over the type of visitors that an expression tree can handle.

```scala
trait Lang {
  trait Visitor[a] { def caseNum(n: int): a }

  type visitor <: Visitor;

  trait Exp {
    def visit[a](v: visitor[a]): a;
  }

  class Num(n: int) extends Exp {
    def visit[a](v: visitor[a]): a = v.caseNum(n);
  }

  class Eval: visitor[Int] extends Visitor[Int] {
    def caseNum(n: int) = n
  }
}
```
trait Lang2 extends Lang {
    
    trait Visitor2[a] extends Visitor[a] { def casePlus(left: Exp, right: Exp): a }

    type visitor <: Visitor2;

    class Plus(l: Exp, r: Exp) extends Exp {
        def visit[a](v: visitor[a]): a = v.casePlus(l, r)
    }

    class Eval2: visitor[Int] extends Eval with Visitor2[Int] {
        def casePlus(l: Exp, r: Exp) = l.visit(this) + r.visit(this)
    }

    class Show2: visitor[String] extends Visitor2[String] {
        def caseNum(n: int) = n.toString()
        def casePlus(l: Exp, r: Exp) = """(" + l.visit(this) + "+" + r.visit(this) + ")"
    }

    "Self typing" : visitor[Int] of Eval necessary so that recursive call is well-typed. (same for Show).
object Main {

    def main(args: Array[String]) = {
        val l1 = new Lang { type visitor = Visitor }
        val e1: l1.Exp = new l1.Num(42);
        System.out.println("eval: "+ e1.visit(new l1.Eval));

        val l2 = new Lang2 { type visitor = Visitor2 }
        System.out.println("eval: "+ e2.visit(new l2.Eval2));
        System.out.println("show: "+ e2.visit(new l2.Show2));
    }
}
Summary

\( \nu\text{Obj} \) is a nominal theory of objects with dependent types.

Nominal means two things:

- The operational semantics uses name passing instead of value passing.
- There are nominal types, whose equality is given “by declaration” instead of “by structure”.

The theory can express among others

- Nominal interface types, as in Java.
- Virtual types and family polymorphism.
- Generative SML structures and functors.

I hope that the theory can be used as a basis for future languages that unify these elements in more flexible and precise constructs for composition and abstraction.
Conclusion

- Interesting component systems will require type systems that go beyond simple universal and existential types.
- We will need to look in more detail at dependent types.
- Much more experience and experimentation is needed.
Appendix: The \( \nu \text{Obj} \) Calculus – Terms

\[x, y, z\] Name
\[l, m, n\] Term label

\[s, t, u \ ::=\] Term
\[x\] Variable
\[t.l\] Selection
\[\nu x \leftarrow t \ ; \ u\] New object
\[\left[x : S \mid \overline{d}\right]\] Class template
\[t \ \&_S \ u\] Composition

\[d \ ::=\] Definition
\[l = t\] Term definition
\[L \leq T\] Type definition

\[p \ ::=\] Path
\[x \mid p.l\]

\[v \ ::=\] Value
\[x \mid \left[x : S \mid \overline{d}\right]\]

roughly corresponds in Scala to:

\[
\text{def} \ x = \text{new} \ t \ ; \ u \\
\text{the value defined by} \ class \ S \ { \overline{d} } \\
\text{tvdb} \ class \ S \ extends \ t \ with \ u
\]
What’s Missing

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

\[
\text{type } S = \{ \text{ type } T; \ldots \} \quad \text{encodes} \quad \text{type } S[T] = \{ \ldots \}.
\]

- Polymorphic functions are encoded as classes with abstract type members.
- In this way the whole of $F_\prec$: can be encoded in $\nu\text{Obj}$.
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 $\nu Obj$

Reduction

\[(select) \quad \nu x \leftarrow [x : S \mid \bar{d}, l = v] ; e\langle x.l \rangle \quad \rightarrow \quad \nu x \leftarrow [x : S \mid \bar{d}, l = v] ; e\langle v \rangle\]
Operational Semantics of $\nu Obj$

Reduction

(select) \hspace{1cm} \nu x \leftarrow [x : S | \overline{d}, l = v] ; e\langle x.l \rangle \rightarrow \nu x \leftarrow [x : S | \overline{d}, l = v] ; e\langle v \rangle

(combine) \hspace{1cm} [x : S_1 | \overline{d}_1] \&_{S} [x : S_2 | \overline{d}_2] \rightarrow [x : S | \overline{d}_1 \oplus \overline{d}_2]
Operational Semantics of $\nu Obj$

Reduction

(select) \[ \nu x \leftarrow [x:S \mid \overline{d}, l = v] ; e(x.l) \rightarrow \nu x \leftarrow [x:S_1 \mid \overline{d}, l = v] ; e\langle v \rangle \]

(combine) \[ [x:S_1 \mid \overline{d}_1] \&_S [x:S_2 \mid \overline{d}_2] \rightarrow [x:S \mid \overline{d}_1 \oplus \overline{d}_2] \]

Structural Equivalence \quad \alpha\text{-renaming of bound variables } x, \text{ plus}

(extrude) \[ e\langle \nu x \leftarrow t ; u \rangle \equiv \nu x \leftarrow t ; e\langle u \rangle \]
Operational Semantics of \( \nu \text{Obj} \)

**Reduction**

\[
\begin{align*}
\text{(select)} & \quad \nu x \leftarrow [x : S \mid \overline{d}, l = v] ; e(x.l) \quad \rightarrow \quad \nu x \leftarrow [x : S \mid \overline{d}, l = v] ; e \langle v \rangle \\
\text{(combine)} & \quad [x : S_1 \mid \overline{d}_1] \&_S [x : S_2 \mid \overline{d}_2] \quad \rightarrow \quad [x : S \mid \overline{d}_1 \uplus \overline{d}_2]
\end{align*}
\]

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

\[
\begin{align*}
\text{(extrude)} & \quad e \langle \nu x \leftarrow t ; u \rangle \quad \equiv \quad \nu x \leftarrow t ; e \langle u \rangle \\
\text{where evaluation context}
\end{align*}
\]

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

• \( \uplus \) is concatenation with overwriting of common labels:

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

• 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:

\[
t \rightarrow u \quad \text{implies} \quad e(t) \rightarrow e(u)
\]

**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 \( t_2 \rightarrow t' \).
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 \( T \).

\[ L \prec: T \] The type label \( L \) represents an **abstract** type
which is bounded by type \( T \).

The right hand side of a \( \prec \) or \( \prec: \) binding can be recursive.

By contrast, recursive aliases are disallowed.
Example: Here’s a simple type definition for lists of integers.

\[
\text{List} \triangleleft \{ \text{isEmpty: Boolean, head: Int, tail: List} \}
\]

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

Two aspects of (\triangleleft):

- \text{List} is a subtype of the \textit{record type}
  \[
  \{ \text{isEmpty: Boolean, head: Int, tail: List} \}.
  \]

- 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

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 \( L \leq: T \).

\( x \) is a name for the object itself (i.e. an \( \alpha \)-renamable version of \textit{this}).

It can be omitted if not referenced.

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

\[
\{ x \mid L = \text{String}; m : x.L \}
\]
**Question:** How does one create a subtype of nominal type?

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

First attempt:

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

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

\[ \text{ListWithLen} < \text{List} & \{ \text{tail: ListWithLen, length: Int} \} \]

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

It has four fields:

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

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 \( T \).
Other Type Constructors

Besides record types and compound types, there are three more type constructors in $\nuObj$.

- A **class type** $[x : S | \overline{D}]$ 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** $p.type$ 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 $p.type\bullet L$. 

38
$\nu$Obj Terms and Types

- $x, y, z$ : Name
- $l, m, n$ : Term label
- $s, t, u ::= $ : Term
  - $x$ : Variable
  - $t.l$ : Selection
  - $\nu x \leftarrow t ; u$ : New object
  - $[x:S | d]$ : Class template
  - $t \&_S u$ : Composition
- $L, M, N$ : Type label
- $S, T, U ::= $ : Type
  - $p \cdot \text{type}$
  - $T \cdot L$
  - $\{x | D\}$
  - $[x:S | D]$ : Class type
  - $T \& U$ : Compound type
- $d ::= $ : Definition
  - $l = t$ : Term definition
  - $L \leq T$ : Type definition
- $D ::= $ : Declaration
  - $l : T$
  - $L \leq: T$
- $p ::= $ : Path
  - $x | p.l$
- $v ::= $ : Value
  - $x | [x:S | d]$
- $\leq:: =$ : Type binder
  - $=$
  - $\prec$
  - $<::$ : Type alias
  - New type
  - Abstract type
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.

\[ \Gamma \vdash T \ni D \]  Type \( T \) contains declaration \( D \).

\[ \Gamma \vdash T < U \]  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)

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

(VarPathI)

\[
\frac{\Gamma \vdash x : R}{\Gamma \vdash x : x\.type}
\]

(SelPathI)

\[
\frac{\Gamma \vdash t : p\.type, \ t.l : R}{\Gamma \vdash t.l : p\.l\.type}
\]

(Sub)

\[
\frac{\Gamma \vdash t : T, \ T \leq U}{\Gamma \vdash t : U}
\]

(New)

\[
\frac{\Gamma \vdash t : [x : S \mid \bar{D}], \ S \prec \{x \mid \bar{D}\} \quad \Gamma, x : S \vdash u : U \quad x \notin \text{fin}(U)}{\Gamma \vdash (\nu x \leftarrow t ; u) : U}
\]

(Class)

\[
\frac{\Gamma \vdash S \text{wf} \quad \Gamma, x : S \vdash \bar{D} \text{wf}, \ t_i : T_i \quad t_i \text{ contractive in } x \quad (i \in 1..n)}{\Gamma \vdash [x : S \mid \bar{D}, l_i = t_i^i \in 1..n] : [x : S \mid \bar{D}, l_i : T_i^i \in 1..n]}
\]

(&)

\[
\frac{\Gamma \vdash t_i : [x : S_i \mid \bar{D}_i] \quad \Gamma \vdash S \text{wf}, \ S \leq S_i \quad (i = 1, 2)}{\Gamma \vdash t_1 \&_S t_2 : [x : S \mid \bar{D}_1 \cup \bar{D}_2]}
\]
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-$\ni$)
  $\Gamma \vdash p\.type \leq \{x \mid \overline{D}', D\}$
  $\Gamma \vdash p\.type \ni [p/x]D$

  (Other-$\ni$)
  $\Gamma, x : T \vdash x\.type \ni D$
  $x \notin \text{fn}(\Gamma, D)$
  $\Gamma \vdash T \ni D$

• For $T$ to contain a declaration, it must be a subtype of a record type.

• If $T$ is a singleton type $p\.type$, 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 $\Gamma \vdash x\.type \ni D$.

• In this case, the resulting $D$ is not allowed to refer to $x$.

42
Properties of $\nu\text{Obj}$

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

Theorem: [Type Soundness] If $\vdash t : T$ then either $t \uparrow$ or $t \rightarrow a$, 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 $F_{<:}$.