The Essence of Dependent Object Types

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, Sandro Stucki

A Long Time Ago in A Galaxy Far Far Away...

A Statically Safe Alternative to Virtual Types

Kim B. Bruce*[1], Martin Odersky[2], and Philip Wadler[3]

[1] Williams College, Williamstown, MA, USA,
kim@cs.williams.edu, http://www.cs.williams.edu/~kim/
[2] University of South Australia,
odersky@cis.unisa.edu.au, http://www.cis.unisa.edu.au/~cismxo/
[3] Bell Labs, Lucent Technologies,
wadler@research.bell-labs.com, http://www.cs.bell-labs.com/~wadler/

Abstract. Parametric types and virtual types have recently been proposed as extensions to Java to support genericity. In this paper we investigate the strengths and weaknesses of each. We suggest a variant of virtual types which has similar expressiveness, but supports safe static

Contents

What was proposed then:

- Languages should have both virtual (abstract) types and type parameters.

What is shown here:

- Virtual types are a great basis for both (and for modules as well).
- Virtual types have a beautiful type theoretic foundation.

Our Aim

We are looking for a *minimal** theory that can model

1. type parameterization,
2. modules,
3. objects and classes.

Our Aim

We are looking for a *minimal** theory that can model

1. type parameterization,
2. modules,
3. objects and classes.

* *minimal*: We do not deal with inheritance; that's deferred to encodings.

Our Aim

We are looking for a *minimal* theory that can model

1. type parameterization,
2. modules,
3. objects and classes.

There were several attempts before, including

> νObj which was proposed as a basis for Scala (ECOOP 2003).

But none of them felt completely canonical or minimal.

Related: 1ML, which can model (1) and (2) by mapping to System F.

Not Everybody Agrees with the Aim

How many FP
people see OOP

How many OOP
people see FP

Dependent Types

We will model *modules* as *objects with type members*.

This requires a notion of dependent type - the type referred to by a type member depends on the owning value.

In Scala we restrict dependencies to *paths*.

In the calculus presented here we restrict it further to *variables*.

Example

We can define *heterogeneous maps* like this:

```
trait Key { type Value }

trait HMap {
  def get(key: Key): Option[key.Value]
  def add(key: Key)(value: key.Value): HMap
}
```

Example

We can define *heterogeneous maps* like this:

```
trait Key { type Value }

trait HMap {
  def get(key: Key): Option[key.Value]
  def add(key: Key)(value: key.Value): HMap
}
```

`type Value` is a *abstract type declaration*
`key.Value` is a *path-dependent type*.

Example

```
trait Key { type Value }

trait HMap {
  def get(key: Key): Option[key.Value]
  def add(key: Key)(value: key.Value): HMap
}

val sort = new Key { type Value = String }
val width = new Key { type Value = Int }

val params = HMap.empty
  .add(width)(120)
  .add(sort)("time")
```

Example

```
trait Key { type Value }

trait HMap {
  def get(key: Key): Option[key.Value]
  def add(key: Key)(value: key.Value): HMap
}

val sort = new Key { type Value = String }
val width = new Key { type Value = Int }

val params = HMap.empty
  .add(width)(120)
  .add(sort)("time")
  .add(width)(true)    // type error
```

Virtual Types can model Type Parameters

Example: Simple Lists in Scala, using type parameters.

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}
```

```
def Nil[T] =
  new List[T] {
    def isEmpty = true
    def head = ???
    def tail = ???
  }
```

```
def Cons[T](hd: T, tl: List[T]) =
  new List[T] {
    def isEmpty = false
    def head = hd
    def tail = tl
  }
```

Encoding using Virtual Types

```
trait List { self =>
  type T
  def isEmpty: Boolean
  def head: T
  def tail: List { type T = self.T }
}
```

```
def Nil[X] =
  new List { self =>
    type T = X
    def isEmpty = true
    def head = self.head
    def tail = self.tail
}
```

```
def Cons[X](hd: X, tl: List { type T = X }) =
  new List { self =>
    type T = X
    def isEmpty = false
    def head = hd
    def tail = tl
  }
```

Covariant Lists

In actual fact, Scala lists are co-variant:

```
trait List[+T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}
```

```
val Nil =
  new List[Nothing] {
    def isEmpty = true
    def head = ???
    def tail = ???
  }
```

```
def Cons[T](hd: T, tl: List[T]) =
  new List[T] {
    def isEmpty = false
    def head = hd
    def tail = tl
  }
```

Encoding Covariance

```
trait List { self =>
  type T
  def isEmpty: Boolean
  def head: T
  def tail: List { type T <: self.T }
}
```

```
val Nil =
  new List { self =>
    type T = Nothing
    def isEmpty = true
    def head = self.head
    def tail = self.tail
}
```

```
def Cons[X](hd: X, tl: List { type T <: X }) =
  new List { self =>
    type T <: X
    def isEmpty = false
    def head = hd
    def tail = tl
  }
```

Encoding Polymorphic Functions

Polymorphic functions can be modeled as dependent functions.

```
trait TypeParam { type TYPE }

def Cons(T: TypeParam)(hd: T.TYPE, tl: List { type T <: T.TYPE }) =
  new List { self =>
    type T < T.TYPE
    def isEmpty = false
    def head = hd
    def tail = tl
  }
```

Towards a Model

What is a maximally simple way to model all this in a calculus?

We need some way to write (dependent) *functions*:

$$\lambda(x : T)t \qquad : \qquad \forall(x : T)U$$

and some way to write *objects*:

$$\nu(x : T)d \qquad : \qquad \mu(x : T)$$

Towards a Model

What is a maximally simple way to model all this in a calculus?

We need some way to write (dependent) *functions*:

$$\lambda(x : T)t \qquad : \qquad \forall(x : T)U$$

and some way to write *objects*:

$$\nu(x : T)d \qquad : \qquad \mu(x : T)$$

Note that all quantifiers range over term variables x.

Objects

An object $\nu(x : T)d$ is composed of a *self reference* $x : T$ and a body d.

The body is composed of *method definitions*:

$$\{a = t\} \quad : \quad \{a : T\}$$

and of *type definitions*:

$$\{A = T\} \quad : \quad \{A : T_1..T_2\}$$

using *aggregation* and *type intersection*:

$$d_1 \wedge d_2 \quad : \quad T_1 \wedge T_2$$

Objects are decomposed using *selection*: $x.a \quad x.A$

Object Types

- The type of an object is a record that can contain self-references.
- Self-references are bound by the recursive type wrapper ν
- For instance, the type of the `List` trait can be modelled like this:

```
List <: μ(self:
            { T: ⊥..⊤  } ∧
            { isEmpty: Boolean } ∧
            { head: self.T } ∧
            { self: List ∧ { T: ⊥..self.T }} )
```

Subtyping

Types are related through subtyping

$$T_1 \quad <: \quad T_2$$

Subtyping is essential, because it gives us a way to relate a path-dependent type $x.A$ to its alias or bounds.

DOT Syntax

x, y, z	**Variable**
a, b, c	**Term member**
A, B, C	**Type member**
$S, T, U ::=$	**Type**
$\{a : T\}$	field declaration
$\{A : S..T\}$	type declaration
$\mu(x{:}T)$	recursive type
$\forall(x{:}S)T$	dependent function
$x.A$	type projection
$S \wedge T$	intersection
$\top$	top type
$\bot$	bottom type

$v ::=$	**Value**
$\nu(x{:}T)d$	object
$\lambda(x{:}T)t$	lambda
$s, t, u ::=$	**Term**
x	variable
v	value
$x.a$	selection
$x\ y$	application
$\textbf{let}\ x = t\ \textbf{in}\ u$	let
$d ::=$	**Definition**
$\{a = t\}$	field def.
$\{A = T\}$	type def.
$d \wedge d'$	aggregate def.

Note: Terms are in ANF form.

This is not a fundamental restriction; it turns out ANF fits well with path-dependent types.

Evaluation

Adopting the techniques of *A Call-By-Need Lambda Calculus*, we define small-step reduction relation using *evaluation contexts* e:

$$
\begin{array}{rcll}
e[t] & \longrightarrow & e[t'] & \textbf{if } t \longrightarrow t' \\
\textbf{let } x = v \textbf{ in } e[x\ y] & \longrightarrow & \textbf{let } x = v \textbf{ in } e[[z := y]t] & \textbf{if } v = \lambda(z\!:\!T)t \\
\textbf{let } x = v \textbf{ in } e[x.a] & \longrightarrow & \textbf{let } x = v \textbf{ in } e[t] & \textbf{if } v = \nu(x\!:\!T)\ldots\{a = t\}\ldots \\
\textbf{let } x = y \textbf{ in } t & \longrightarrow & [x := y]t & \\
\textbf{let } x = \textbf{let } y = s \textbf{ in } t \textbf{ in } u & \longrightarrow & \textbf{let } y = s \textbf{ in let } x = t \textbf{ in } u &
\end{array}
$$

$$\textbf{where } e ::= [\,] \mid \textbf{let } x = [\,] \textbf{ in } t \mid \textbf{let } x = v \textbf{ in } e$$

Type Assignment

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

$$\frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda(x{:}T)t : \forall(x{:}T)U}$$

$$\frac{\Gamma \vdash x : \forall(z{:}S)T \qquad \Gamma \vdash y : S}{\Gamma \vdash x\, y : [z := y]T}$$

$$\frac{\Gamma, x : T \vdash d : T}{\Gamma \vdash \nu(x{:}T)d : \mu(x{:}T)}$$

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T}$$

$$\frac{\begin{array}{c} x \notin fv(U) \\ \Gamma \vdash t : T \qquad \Gamma, x : T \vdash u : U \end{array}}{\Gamma \vdash \mathbf{let}\ x = t\ \mathbf{in}\ u : U}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x{:}T)}$$

$$\frac{\Gamma \vdash x : \mu(x{:}T)}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U}$$

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

Type Assignment

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

$$\frac{x \notin \mathit{fv}(U) \quad \Gamma \vdash t : T \qquad \Gamma,\, x : T \vdash u : U}{\Gamma \vdash \mathbf{let}\ x = t\ \mathbf{in}\ u : U}$$

$$\frac{\Gamma,\, x : T \vdash t : U}{\Gamma \vdash \lambda(x{:}T)t : \forall(x{:}T)U}\ (\forall\text{-I})$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x{:}T)}$$

$$\frac{\Gamma \vdash x : \forall(z{:}S)T \qquad \Gamma \vdash y : S}{\Gamma \vdash x\, y : [z := y]T}\ (\forall\text{-E})$$

$$\frac{\Gamma \vdash x : \mu(x{:}T)}{\Gamma \vdash x : T}$$

$$\frac{\Gamma,\, x : T \vdash d : T}{\Gamma \vdash \nu(x{:}T)d : \mu(x{:}T)}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U}$$

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T}$$

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

Type Assignment

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

$$\frac{x \notin fv(U) \quad \Gamma \vdash t : T \qquad \Gamma,\, x : T \vdash u : U}{\Gamma \vdash \mathbf{let}\ x = t\ \mathbf{in}\ u : U}$$

$$\frac{\Gamma,\, x : T \vdash t : U}{\Gamma \vdash \lambda(x{:}T)t : \forall(x{:}T)U}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x{:}T)}$$

$$\frac{\Gamma \vdash x : \forall(z{:}S)T \qquad \Gamma \vdash y : S}{\Gamma \vdash x\, y : [z := y]T}$$

$$\frac{\Gamma \vdash x : \mu(x{:}T)}{\Gamma \vdash x : T}$$

$$\frac{\Gamma,\, x : T \vdash d : T}{\Gamma \vdash \nu(x{:}T)d : \mu(x{:}T)} \ (\textsc{New})$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U}$$

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \ (\textsc{Sel})$$

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

Type Assignment

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

$$\frac{\Gamma,\ x : T \vdash t : U}{\Gamma \vdash \lambda(x{:}T)t : \forall(x{:}T)U}$$

$$\frac{\Gamma \vdash x : \forall(z{:}S)T \qquad \Gamma \vdash y : S}{\Gamma \vdash x\ y : [z := y]T}$$

$$\frac{\Gamma,\ x : T \vdash d : T}{\Gamma \vdash \nu(x{:}T)d : \mu(x{:}T)}$$

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T}$$

$$\frac{\begin{array}{c} x \notin fv(U) \\ \Gamma \vdash t : T \qquad \Gamma,\ x : T \vdash u : U \end{array}}{\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ u : U}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x{:}T)} \quad (\textsc{Rec-I})$$

$$\frac{\Gamma \vdash x : \mu(x{:}T)}{\Gamma \vdash x : T} \quad (\textsc{Rec-E})$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \quad (\wedge\text{-I})$$

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

Type Assignment

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

$$\frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda(x{:}T)t : \forall(x{:}T)U}$$

$$\frac{\Gamma \vdash x : \forall(z{:}S)T \qquad \Gamma \vdash y : S}{\Gamma \vdash x\,y : [z := y]T}$$

$$\frac{\Gamma, x : T \vdash d : T}{\Gamma \vdash \nu(x{:}T)d : \mu(x{:}T)}$$

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T}$$

$$\frac{\begin{array}{c} x \notin \mathrm{fv}(U) \\ \Gamma \vdash t : T \qquad \Gamma, x : T \vdash u : U \end{array}}{\Gamma \vdash \textbf{let } x = t \textbf{ in } u : U}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x{:}T)}$$

$$\frac{\Gamma \vdash x : \mu(x{:}T)}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: U}{\Gamma \vdash t : U}\ (\textsc{Sub})$$

Definition Type Assignment

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} : \{a : T\}}$$

$$\Gamma \vdash \{A = T\} : \{A : T..T\}$$

$$\frac{\begin{array}{c} dom(d_1) \cap dom(d_2) = \emptyset \\ \Gamma \vdash d_1 : T_1 \qquad \Gamma \vdash d_2 : T_2 \end{array}}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2}$$

Subtyping

$$\Gamma \vdash T <: T$$

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

$$\Gamma \vdash T <: \top$$

$$\Gamma \vdash \bot <: T$$

$$\Gamma \vdash T \wedge U <: T$$

$$\Gamma \vdash T \wedge U <: U$$

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

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T}$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A}$$

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

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma,\, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : S_1)T_1 <: \forall(x : S_2)T_2}$$

Subtyping

$$\Gamma \vdash T <: T$$

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

$$\Gamma \vdash T <: \top$$

$$\Gamma \vdash \bot <: T$$

$$\Gamma \vdash T \wedge U <: T$$

$$\Gamma \vdash T \wedge U <: U$$

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

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \; (\text{TSel-}<:)$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \; (<:\text{-TSel})$$

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

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma,\, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : S_1) T_1 <: \forall(x : S_2) T_2}$$

Meta-Theory

Simple as it is, the soundness proof of DOT was surprisingly hard.

- Attempts were made since about 2008.
- Previous publications (FOOL 12, OOPSLA 14) report about (some) advances and (lots of) difficulties.
- Essential challenge: Subtyping theories are *programmer-definable*.

Programmer-Definable Theories

In Scala and DOT, the subtyping relation is given in part by user-definable definitions:

```
type T >: S <: U                    { T: S .. U }
```

This makes `T` a supertype of `S` and a subtype of `U`.

By transitivity, `S <: U`.

So the type definition above proves a subtype relationship which was potentially not provable before.

Bad Bounds

What if the bounds are non-sensical?

Example

```
type T >: Any <: Nothing
```

By the same argument as before, this implies that

```
Any <: Nothing
```

Once we have that, again by transitivity we get $S <: T$ for arbitrary S and T.

That is, the subtyping relations collapses to a single point.

Bad Bounds and Inversion

A collapsed subtyping relation means that inversion fails.

Example: Say we have a binding $x = \nu(x\colon T)\ldots$.

So in the corresponding environment Γ we would expect a binding $x : \mu(x\colon T)$.

But if every type is a subtype of every other type, we also get with subsumption that $\Gamma \vdash x : \forall(x : S)U$!

Hence, we cannot draw any conclusions from the type of x. Even if it is a function type, the actual value may still be a record.

Can We Exclude Bad Bounds Statically?

Unfortunately, no.

Consider:

```
type S = { type A; type B >: A <: Bot }
type T = { type A >: Top <: B; type B }
```

Individually, both types have good bounds. But their intersection does not:

```
type S & T == { type A >: Top <: Bot; type B >: Top <: Bot }
```

So, bad bounds can arise from intersecting types with good bounds.

It turns out that even checking all intersections of a program statically would not exclude bad bounds.

Dealing With It

Observation: To prove preservation, we need to reason at the top-level only about environments that arise from an actual computation. I.e. in

If $\Gamma \vdash t : T$ and $t \longrightarrow u$ then $\Gamma \vdash u : T$.

the environment Γ corresponds to an evaluated `let` prefix, which binds variables to values.

And values have guaranteed good bounds because all type members are aliases.

$$\Gamma \vdash \{A = T\} : \{A : T..T\}$$

The paper provides an elaborate argument how to make use of this observation for the full soundness proofs.

Variants

- First soundness proof by Tiark and Nada used big-step semantics for a variant of DOT.
- That variant is more powerful (and its meta-theory more complicated) because it deals with subtyping recursive types.
- The paper presents an independently developed proof that uses a small-step semantics.
- We took heed of Phil's advice of the importance of being stupid.

Conclusion