Dependent Object Types

EDIC Candidacy Exam

Nada Amin

LAMP, I&C, EPFL

September 12, 2012

DOT: Dependent Object Types

The DOT calculus proposes a new *type-theoretic foundation* for Scala and languages like it. It models

- *path-dependent types*
- abstract type members
- mixture of nominal and structural typing via refinement types

It does not model

- inheritance and mixin composition
- what's currently in Scala

Path-dependent types

path-dependent type
: limited form of *dependent type*, in which a type depends on a *path*

dependent type
: a type which depends on a term

path
: a chain of immutable fields or stable values

```
abstract class AbsCell {
  type T
  val init: T
  private var value: T = init
  def get = value
  def set(x: T) { value = x }
}
```

```
val c = new AbsCell {
  type T = Int
  val init = 1
}
c.set(2)
val a: AbsCell = c
// a.T is opaque
a.set(a.init)
```

```
object Library {
  def update(c: AbsCell)(oldval: c.T, newval: c.T) =
    if (oldval == c.get) { c.set(newval); true }
    else false
}
```

Selected Papers

1. *A Type-Theoretic Approach to Higher-Order Modules with Sharing*
   by Robert Harper and Mark Lillibridge
   POPL '94
   - calculus for modules in the ML tradition
   - modules are first-class values
2. *Tribe: a simple virtual class calculus*
   by Dave Clarke, Sophia Drossopoulou, James Noble and Tobias Wrigstad
   AOSD '07
   - calculus for virtual classes
   - paths are types, and types are "generalized" paths
3. *A Nominal Theory of Objects with Dependent Types*
   by Martin Odersky, Vincent Cremet, Christine Röckl and Matthias Zenger
   ECOOP '03
   - calculus unifying advanced object-oriented and module systems
   - foundation for Scala

SML modules: Structures and Signatures

```
structure A = struct                  (* A.T transparent *)
  type T = int                        val vA: int = A.v
  val v = 0
  val f = fn x => x + 1
end

signature X = sig
  type T
  val v: T
  val f: T -> T
end

structure B :> X = A                  (* B.T opaque *)
                                      val vB: B.T = B.v
```

SML modules: Functors and Type Sharing

```
functor mkDoubler(arg: X) :> X where type T = arg.T = struct
  type T = arg.T
  val v = arg.v
  val f = arg.f o arg.f
end

(* A2.T = A.T = int *)
structure A2 = mkDoubler(A);
val a: int = A2.f(A.v);

(* B2.T = B.T *)
structure B2 = mkDoubler(B);
val b: B.T = B2.f(B.v);
```

SML modules: Recap

structure a module
a collection of types, values and (sub)structures packaged as a program unit.

signature an interface; the type of a structure
description of the types, values and (sub)structures of a structure given by their kinds, types and interfaces.

functor a function mapping structures to structures

type-sharing enables stating that types specified in separate modules are actually equal

Limitations of SML modules

- modules are not first-class; e.g. not possible to return a structure from an `if` expression
- type sharing is restricted to equality between type names, not general type expressions (limitation of SML '90; lifted in SML '97)

  ```
  signature X' = sig
    type T
    type T2T = T -> T
    val v: T
    val f: T2T
  end
  ```

- in effect, transparency / opaqueness of a signature cannot be fully fined-tuned

"Translucent" ML: Calculus

grounded in type theory; based on Girard's F_ω, adding:

- *translucent sums* to model modules
- dependent functions to model functors
- a notion of subtyping to model module implementation-interface matching

"Translucent" ML: Translucent Sums

- translucent sums are values representing modules
- translucent sum: sequence of bindings
- translucent sum type: sequence of declarations
- unlike traditional records, later fields can depend on earlier ones
- fields can be types or terms
- any type can be partially or fully determined by a type expression
- dependent typing since translucent sums are terms that may contain type components

"Translucent" ML: Example

```
structure XInt = struct
  type T = int
  val v = 3
  val f = negate
end
structure XBool = struct
  type T = bool
  val v = true
  val f = not
end
```

```
signature XType = sig
  type T
  val v: T
  val f: T -> T
end
```

```
structure X =
  if flip() then XInt else XBool
```

“Translucent” ML: Path-Dependent Types

- Scala’s abstract types $\approx$ ML’s abstract types of signatures; limitations:
  - recursive references
  - bounded quantifications
- translucent sum can be given a more precise type by referring to the name of its type member with a path selection; essential for
  - breaking the dependencies between (sub)fields
  - propagating typing information

Tribe: Example (Virtual Classes)

```
class Graph {
  class Node {
    Edge connect(Node other) {
      return new Edge(this, other);
    }
  }
  class Edge {
    Node from, to;
    Edge(Node f, Node t) { from = f; to = t; }
  }
}

class ColouredGraph extends Graph { // subclassing
  class Node {                      // further binding
    Colour nodeColour;
  }
}
```

Tribe: Example (Object Family and Family Polymorphism)

```
final ColouredGraph cg1, cg2;
cg1.Node cn1, cn3;
cg2.Node cn2;
cn1.connect(cn3); // Type Correct
cn2.connect(cn3); // Type Error!!!

class Library {
  int distance(Graph.Node n1, n1.out.Node n2)  { ... }
  int distance2(Graph g, g.Node n1, g.Node n2) { ... }

  e.out.Edge copyEdge(Graph.Edge e) {
    e.out.Node from = e.from;
    e.out.Node to = e.to;
    new e.out.Edge(from, to);
  }
}
```

Tribe Types

- in Tribe, types are a form of generalized paths: a final variable, `this` or a class name then followed by a possibly empty sequence of field, out or class selections
- `kitt.Passenger.name`: the name of one of Kitt's passengers
- `kitt.driver <: Car.driver <: Car.Passenger <: Car.Traveller <: Vehicle.Traveller`
- `kitt.driver` $\not<:$ `karr.driver` and `kitt.Passenger` $\not<:$ `karr.Passenger`

```
class Vehicle { class Traveller { ... } }
class Car extends Vehicle {
  class Passenger extends Traveller {
    class String { ... }
    final String name;
  }
  final Passenger driver;
}
final Car kitt;
final Car karr;
```

Tribe Calculus: Recap

- *families* of classes inherited together
- classes are lexically nested inside other classes: when a class is inherited, its nested inner classes are inherited along with their methods and fields
- two form of inheritance: subclassing and further binding
- *family polymorphism*: code written for one family also works for extensions of that family
- in Tribe:
  - path types can depend simultaneously on both classes and objects
  - paths can use an `out` field to move from an object to the object which surrounds it; enables ubiquitous access to an object's family without the need to drag around family arguments

Tribe: Discussion

- no virtual classes in Scala; its virtual types can emulate some (but not all) of the benefits
- because of `out` field, Tribe has limited cross-family inheritance

```
class A {
  class D { ... }
  class B {
    class C {
      this.out.out.D foo;
    }
  }
}
class E {
  class F extends A.B.C { ... }
}
```

- soundness? substitution lemma seems wrong when `null` is substituted for a variable in an expression whose type contains that variable
- aims to have decidable type-checking, but still not proved so

νObj

A calculus for classes and objects which can have types as members. It can encode:

- Java's inner classes
- virtual types
- family polymorphism
- essential aspects of ML-like module systems, including
  - sharing constraints
  - higher-order functors

A basis for unifying concepts in advanced object and module systems. Connections:

- Object = Module
- Object type = Signature
- Class = Method = Functor

νObj: Syntax

- terms in νObj denote objects or classes; consist of:

variables	x
selections	$t.l$
object creations	$\nu x \leftarrow t \; ; \; u$
class templates	$[x : S \mid \overline{d}]$
mixin compositions	$t_1 \;\&_S\; t_2$
	(if omitted, $S = S_1 \;\&\; (S_2 \;\&\; \{x \mid D_1 \uplus D_2\})$)

- a value is a variable or class template
- a path is a variable followed by a possibly empty sequence of selections
- types in νObj consist of:

singleton types	$p.\textbf{type}$
type selections	$T \bullet L$
record types	$\{x \mid \overline{D}\}$
class types	$[x : S \mid \overline{D}]$
compound types	$T \;\&\; U$

νObj: Details

- type bindings

  type aliases $=$
  new types $\prec$
  abstract types $<:$
- class template $[x : S|\ \overline{d}]$ related to translucent sum; some differences:
  - needs to be instantiated
  - "contractive" restriction
- class type $[x : S|\ \overline{D}]$
  - contains as values classes that instantiate to objects of (sub)type $\{x|\ \overline{D}\}$
  - explicit self type S may be different from $\{x|\ \overline{D}\}$
  - declarations in S which are not in $\overline{D}$ play the role of abstract members, defined by mixin composition during instantiation

νObj: Example (encoding of monomorphic functions)

- Encoding for a λ-abstraction $\lambda(x : T)\ t$:

  $$[\mathtt{x}:\ \{\mathtt{arg}:\ \mathtt{T}\}|\ \mathtt{fun} = [\mathtt{res} = t']]$$

  ($t' = t$ with `x.arg` substituted for x)

- Encoding for an application $g(e)$:

  $$\nu\, \mathtt{g}_{app} \leftarrow \mathtt{g}\ \&\ [\mathtt{arg} = \mathtt{e}];$$
  $$\nu\, \mathtt{g}_{eval} \leftarrow \mathtt{g}_{app}.\mathtt{fun};$$
  $$\mathtt{g}_{eval}.\mathtt{res}$$

νObj: Discussion

- conflate the concepts of compound types (which inherit the members of several parent types) and mixin composition (which build classes from other classes and traits)
- mixin composition is not commutative, unlike classical intersection types
- in Scala, least upper bounds and greatest lower bounds do not always exist, e.g:

```
trait A { type T <: A }
trait B { type T <: B }
// glb is an infinite sequence
A with B { type T <: A with B {
  type T <: ...
}}}
```

DOT: Dependent Object Types

- core calculus for modeling *path-dependent types*
- we've seen path-dependent types arise in three settings:
  - ML-like module systems
  - virtual classes
  - ν*Obj* / Scala
- DOT aims to bring more uniformity and simplicity to Scala
  - replace Scala's compound types with classical intersection types
  - complement calculus with classical union types
  - intersections and unions form a lattice wrt subtyping

DOT: Status

- basic calculus
- type safety
  - showed by counterexamples that subject reduction doesn't hold
  - sketch of plausible proof via logical relations
- submission about this work-in-progress accepted to Foundations of Object-Oriented Languages (FOOL '12)
- future work
  - confirm proof of type safety and/or refine calculus
  - investigate translations of Scala features into DOT
  - implement a compiler front-end which uses DOT for typechecking

DOT: Syntax

- terms

variables	x, y, z
selections	$t.l$
method invocations	$t.m(t)$
object creations	$\textbf{val } y = \textbf{new } c \;\; t'$
	c is a constructor $T_c \left\{ \overline{l = v} \; \overline{m(x) = t} \right\}$

- types

type selections	$p.L$
refinement types	$T \left\{ z \Rightarrow \overline{D} \right\}$
type intersections	$T \wedge T'$
type unions	$T \vee T'$
a top type	$\top$
a bottom type	$\bot$

DOT: Preservation Counterex. (Well-Formedness Lost)

// $y.A$ is not well-formed after v is substituted for x.

$$
\begin{array}{l}
\textbf{val } v = \textbf{new } \top \{z \Rightarrow L : \bot..\top \{z \Rightarrow A : \bot..\top, B : z.A..z.A\}\} \{\} \\
(\textbf{app } \lambda x{:}\top \{z \Rightarrow L : \bot..\top \{z \Rightarrow A : \bot..\top, B : \bot..\top\}\} . \\
\quad \textbf{val } z = \textbf{new } \top \{z \Rightarrow l : \bot \to \top\} \{ \\
\quad\quad l = \lambda y{:}x.L \wedge \top \{z \Rightarrow A : z.B..z.B, B : \bot..\top\}. \\
\quad\quad\quad \lambda a{:}y.A.(\textbf{app } (\lambda x{:}\top.x)\ a)\} \\
\ (\textbf{app } (\lambda x{:}\top.x)\ z) \\
\ v)
\end{array}
$$

System F_ω: Higher-order polymorphic lambda-calculus

terms t	typed lambda-calculus terms	$x - \lambda x : T.t - t\ t$
	type abstraction	$\lambda X :: K.t$
	type application	$t[T]$
types T	type variable	X
	type of functions	$T \to T$
	universal type	$\forall X :: K.T$
	operator abstraction	$\lambda X :: K.T$
	operator application	$T\ T$
kinds K	kind of proper types	$*$
	kind of operators	$K \Rightarrow K$

System $F_{<:}$: Bounded Quantification

terms t	typed lambda-calculus terms	x – $\lambda x : T.t$ – $t\ t$
	type abstraction	$\lambda X <: T.t$
	type application	$t[T]$
types T	type variable	X
	maximum type	$\top$
	type of functions	$T \to T$
	universal type	$\forall X <: T.T$