Dependent Object Types

Nada Amin
LAMP, I&C, EPFL

***Abstract*—Our aim is to propose a new type-theoretic foundation of Scala based on a core calculus for path-dependent types. Towards this end, we review three papers presenting calculi in which path-dependent types are essential: a calculus for first-class translucent ML-like modules, the Tribe calculus for virtual classes, and the νObj calculus for reasoning about classes and objects with type members.**

***Index Terms*—calculus, objects, dependent types, scala**

I. Introduction

THE aim of this research is to propose a new sound and tractable type-theoretic foundation of Scala [1] and languages like it: the Dependent Object Types calculus (DOT). The properties we are interested in modeling are Scala's path-dependent types and abstract type members, as well as its mixture of nominal and structural typing through the use of refinement types. Compared to previous approaches [2], [3], we make no attempt to model inheritance or mixin composition. Furthermore, the calculus is more normative than descriptive, in that it does not precisely describe what's currently in Scala.

Path-dependent types are a key concept in the DOT calculus, and a unifying concept of the three papers we choose to study. A *dependent type* is a type which depends on a term. A *path-dependent type* is a limited form of dependent type, in which a type depends on a path: a chain of immutable fields or stable values. Path-dependent types arise when terms, usually objects, can have types as members. Abstract type members, together with explicit selftypes and modular mixin composition, have been identified as key abstractions for the construction of reusable components [4].

Proposal submitted to committee: September 5th, 2012; Candidacy exam date: September 12th, 2012; Candidacy exam committee: Rachid Guerraoui, Martin Odersky, Viktor Kuncak.

Scala Example: In the code below, `type T` is an abstract type member of `AbsCell`. In a library method taking an abstract cell `c`, the expression `c.set(c.init)` is well-typed because `c.init` has type `c.T` and `c.set` has the type of a function that expects a parameter of type `c.T`. Note that `c.T` is a path-dependent type.

```
abstract class AbsCell {
  type T
  val init: T
  private var value: T = init
  def get = value
  def set(x: T) { value = x }
}
object Library {
  def reset(c: AbsCell) { c.set(c.init) }
  def update(c: AbsCell)(oldval: c.T, newval: c.T) =
    if (oldval == c.get) { c.set(newval); true }
    else false
}
```

In section II, we survey selected previous work related to path-dependent types. Whenever possible, we contrast the formalisms with concrete examples in Scala. In section III, we discuss in more details our proposal for a new type-theoretic foundation of Scala based on path-dependent types.

II. Survey of the Selected Papers

We present three papers [5], [6], [2], each of which models a calculus with path-dependent types.

- The first paper, *A Type-Theoretic Approach to Higher-Order Modules with Sharing* by Robert Harper and Mark Lillibridge [5], presents a calculus for modules in the ML tradition. Path-dependent types arise because modules, which may contain type components, are first-class values in their calculus.
- The second paper, *Tribe: a simple virtual class calculus* by Dave Clarke, Sophia Drossopoulou, James Noble and Tobias Wrigstad [6], presents a core calculus for virtual classes. Path-dependent types arise because, in Tribe, paths are types, and, almost vice versa: each type is a generalized form of path.
- The third paper, *A Nominal Theory of Objects with Dependent Types* by Martin Odersky, Vincent Cremet, Christine Röckl and Matthias Zenger [2], presents a calculus which unifies concepts from advanced object-oriented systems and module systems and, with [3], serves as a type-theoretic foundation for many features of Scala's type system. Path-dependent types arise because any path p has a singleton type $p.$**type**, from which type members can be selected.

A. A Type-Theoretic Approach to Higher-Order Modules with Sharing

In [5], Harper and Lillibridge present a calculus for first-class modules, with complete control over the propagation of information between compile-time program units.

Their module system is inspired by Standard ML (SML). The SML module system has three constructs: *signatures*, *structures*, and *functors*. A *structure* is a module: it defines a collection of types, values and (sub)structures packaged as a program unit. A *signature* is an interface, the "type" of a structure: it describes the types, values and (sub)structures of a structure by giving their kinds, types and interfaces. A *functor* is a function mapping structures to structures. SML has a notion of "type sharing" which allows stating that types specified in separate modules are actually equal.

Their module system addresses some shortcomings of the SML module system. Though the SML module system is higher-order, because SML provides functors and substructures, it still treats modules as second-class. For example, it isn't possible to return a structure from an `if` expression. In fact, allowing this might require obscuring the visibility of the resulting structure to ensure soundness, which we illustrate with an example below. Another limitation of the SML module system is that "type sharing" is restricted to equality between type names, not general type expressions. In effect, this means that the transparency / opaqueness of a signature cannot be fully fine-tuned. Their proposal introduces translucent signatures, which allows any type to be partially or fully determined by a type expression.

Their calculus is grounded in type theory. It is based on Girard's F_ω [7]. They add *translucent sums* to model modules, dependent functions to model functors, a notion of subtyping to model module implementation-interface matching. The crucial addition and novelty is that of a *translucent sum*. A translucent sum has the form of a possibly empty sequence of bindings. Similarly, a translucent sum type has the form of a possibly empty sequence of declarations. They differ from traditional records, because they can contain types and because later fields can depend on earlier ones. Translucent sums are ordinary values. Since they model modules, it means that modules are first-class values, and their module system is higher-order "for free".

Scala's abstract types have close resemblances to abstract types of signatures in these ML-like module systems. In particular, in their calculus, there's a form of dependent type, since translucent sums are terms that may contain types. More concretely, their so-called VALUE rules enable a translucent sum to be given a more precise type by referring to the names of its type members with a path selection – this ability based on path-dependent types is not only essential for breaking the dependencies between (sub)fields but also critical for the propagation of typing information. Compared to Scala, these path-dependent types are more limited because recursive references and bounded quantifications are not allowed.

In terms of theoretical results, Lillibridge shows in his thesis that a simplified kernel system based on this calculus is sound, even in the presence of side effects [8]. In the appendix of the paper, they show that subtyping, and hence type checking, is undecidable.

ML-like Example and Scala Encoding: Consider this example, in a ML-like language implementing the calculus of the paper. Since modules are first-class, we can `flip` a coin and return a module depending on the outcome:

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

But now, what should be the static type or signature of structure `X`? In ML, in order to make the `if` expression typecheck, the two branches must have equal types. In the calculus, the subsumption rule is used to coerce both types to their least upper bound, represented by the signature:

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

It is straightforward to encode this example in Scala, with the correspondence Object = Module, and Object Type = Signature. Indeed, the νObj calculus, which we study in section II-C embraces this correspondence, and further claims: Class = Method = Functor, as we will see. Note that type members, and hence path-dependent types, are crucial for this correspondence.

```
object Test extends App {
  object coin {
    val r = new util.Random()
    def flip() = r.nextBoolean()
  }
  type XType = {
    type T
    val v: T
    val f: T ⇒ T
  }
  val xInt = new {
    type T = Int
    val v = 3
    val f = - (_:T)
  }
  val xBool = new {
    type T = Boolean
    val v = true
    val f = ! (_:T)
  }
  val x: XType = if (coin.flip()) xInt else xBool

  println(x.f(x.v))
}
```

B. Tribe: a simple virtual class calculus

In [6], Clarke et al. present a calculus for virtual classes. Virtual classes allow *families* of classes to be inherited, rather than just single classes. Classes are lexically nested inside other classes. When a class is inherited, its nested inner classes are inherited along with their methods and fields. *Further binding* enables the nested classes to be further extended. Path-dependent types arise naturally to distinguish objects coming from different families. Yet, thanks to *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. Additionally, paths can use an `out` field to move from an object to the object which surrounds it. This `out` feature enables ubiquitous access to an object's family without the need to drag around family arguments.

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. Given the code below, we can read the type `kitt.Passenger.name` as the name of one of Kitt's passengers (including the driver). Subtyping is natural: `kitt.driver` <: `Car.driver` <: `Car.Passenger` <: `Car.Traveller` <: `Vehicle.Traveller`. Crucially, `kitt.driver` is not a subtype of `karr.driver` and similarly for `kitt.Passenger` and `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;
```

Though Scala doesn't have virtual classes, its abstract types enable it to emulate some (but not all) of the benefits of virtual classes as presented in the Tribe calculus. On the other hand, Tribe doesn't have virtual types. In a follow-up paper [9], an improved variant of Tribe with simpler types, cross-family inheritance and generics is presented and a formalization of ownership types built upon it.

In terms of theoretical results, there's a sketch of soundness in the paper, but it's unclear whether it is correct. In particular, the substitution lemma does not seem valid when `null` is the value to be substituted and the expression has a type which contains the variable to be substituted. In the follow-up paper [9], the soundness proof sketch seems more plausible but then is tied to the ownership system formalism. Decidability of subtyping, and hence typechecking, has not yet been established.

Tribe Example and Scala Encoding: Let's now re-use an example from the paper, and discuss how we would emulate it in Scala.

```
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;
  }
}

class Library {
  int distance(Graph.Node n1, n1.out.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);
  }
}
```

The "tests" are:

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

Now, we can emulate this Tribe example in Scala at the expense of some syntactic overhead, since further binding and `out` selection are not native. We use virtual types to ensure that the inner classes are indeed "virtual", i.e. dynamically instead of statically dispatched. In order to conform to the "virtual class" pattern, the leaf classes `Graph` and `ColouredGraph` are no longer in a subclassing / further-binding with one another, as this would only bring confusion when ascribing a coloured graph to a graph, since the constructors are statically dispatched. Emulating `out` faithfully involves a bit of hackery in making Scala recognize that, for example, `e.from.type` is a subtype of `e.out.Node`. As noted in the Tribe paper, using `out` in library code obscures the symmetry between the parameters (for example, in the `distance` method below) but allows the family object to be recovered without being explicitly passed around (contrast with the method `distance2`).

```
trait GraphBase { graph ⇒
  type Node <: NodeBase
  type Edge <: EdgeBase
  def newNode(): Node
  def newEdge(from: Node, to: Node): Edge

  trait WithOut {
    val out: graph.type = graph
  }
  trait NodeBase extends WithOut { this: Node ⇒
    def connect(other: out.Node): out.Edge =
      newEdge(this, other)
  }
  trait EdgeBase extends WithOut { this: Edge ⇒
    val from: out.Node
    val to: out.Node
  }
}

class Graph extends GraphBase {
  class Node extends NodeBase
  class Edge(val from: Node, val to: Node) extends EdgeBase
  def newNode() = new Node
  def newEdge(from: Node, to: Node) = new Edge(from, to)
}

class ColouredGraph extends GraphBase {
  class Node(val colour: String) extends NodeBase
  class Edge(val from: Node, val to: Node) extends EdgeBase
  def newNode() = new Node("blue")
  def newEdge(from: Node, to: Node) = new Edge(from, to)
}

object Library {
  def distance(n1: GraphBase#Node)(n2: n1.out.Node): Int = {
    val e: n1.out.Edge = n1.connect(n2)
    0 // ...
  }
  def copyEdge(e: GraphBase#Edge): e.out.Edge =
    e.out.newEdge(e.from, e.to)

  def distance2(g: GraphBase)(n1: g.Node, n2: g.Node): Int = {
    val e: g.Edge = n1.connect(n2)
    0 // ...
  }
  def copyEdge2(g: GraphBase)(e: g.Edge): g.Edge =
    g.newEdge(e.from, e.to)
}
```

```
object Test extends App {
  val cg1, cg2 = new ColouredGraph()
  val cn1 = new cg1.Node("blue")
  val cn2 = new cg2.Node("green")
  val cn3 = new cg1.Node("red")
  val e = cn1.connect(cn3)
  // this is a type mismatch
  // cn2.connect(cn3)

  val d1 = Library.distance(cn1)(cn3)
  val d2 = Library.distance2(cg1)(cn1, cn3)
  val e1 = Library.copyEdge(e)
  val e2 = Library.copyEdge2(cg1)(e)
  // these are all type mismatches
  // Library.distance(cn1)(cn2)
  // Library.distance2(cg2)(cn1, cn3)
  // Library.copyEdge2(cg2)(e)
}
```

C. A Nominal Theory of Objects with Dependent Types

In [2], Odersky et al. presents νObj, a calculus for classes and objects which can have types as members. It proposes three kinds of type members: aliases, abstract types, and new types. The calculus can encode Java's inner classes, virtual types, family polymorphism, essential aspects of the ML-like module systems including sharing constraints and higher-order functors, and System $F_{<:}$ [7]. The calculus can be used as a basis for unifying concepts that so far existed in parallel in advanced object systems and module systems. νObj makes the following connections: Object = Module, Object type = Signature, Class = Method = Functor.

The terms in νObj denote objects or classes. They consist of variables x denoting objects, selections $t.l$, object creations $\nu x \leftarrow t \,;\, u$ defining an object x of class t whose scope is the term u, class templates $[x : S|\, \overline{d}]$, and mixin compositions $t \,\&_S\, u$ denoting a class of self type S combined from two classes t and u. A value is a variable or a class template. A path is a variable followed by a possibly empty sequence of selections.

The types in νObj consist of singleton types $p.$**type**, type selections $T\bullet L$, record types $\{x|\, \overline{D}\}$, compound types $T \,\&\, U$, and class types $[x : S|\, \overline{D}]$.

A class template $[x : S|\, \overline{d}]$ is closely related to a translucent sum of section II-A, except that it needs to be instantiated into an object while a translucent sum is a module. Like in a translucent sum, $\overline{d}$ binds term labels to values and type labels to types. The x of type S stands for the self, i.e. the object being constructed from the template. Its scope is the bindings $\overline{d}$. Since the self is explicitly typed, the bindings in class templates can "contractively" refer to each other (and even to themselves) via x, while recall that, in translucent sums, only later fields can refer to earlier ones but the references don't have any "contractive" restrictions. Roughly, "contractiveness" in νObj guarantees that a field cannot be referenced before it is initialized.

Contrasting concrete type bindings $\overline{d}$ used in class template terms with type declarations $\overline{D}$ used in record and class types, the possible type binders in type declarations are type aliases $=$, new types $\prec$ and abstract types $<:$. Only the first two are also concrete type binders available in type bindings of class templates.

A class type $[x : S|\, \overline{D}]$ contains as values classes that instantiate to objects of (sub)type $\{x|\, \overline{D}\}$. The 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 – they must be defined by mixin composition in order for the class to be instantiated.

Mixin composition is not commutative. When expanding a type to a record type, a compound type $R_1 \,\&\, R_2$ expands to a record type containing the concatenation of the declarations in R_1 and R_2, but if some label is defined in both R_1 and R_2, the definition in R_2 overrides (and so must be more specific than) the definition in R_1. Though this break of commutativity is justified from an implementation standpoint, it is awkward from a typing standpoint: in particular, least upper bounds and greatest lower bounds do not always exist in Scala.

In terms of theoretical results, the paper confirms the type soundness of the calculus and the undecidability of its type checking by reduction to $F_{<:}$.

νObj Example: We show how to encode monomorphic functions into the calculus. The paper goes further and generalizes the encoding to system $F_{<:}$.

A λ-abstraction $\lambda(x : T)\ t$ is represented as a class with an abstract member `arg` for the function argument:

```
[x: {arg: T}| fun = [res = t′]]
```

t' corresponds to term t in which all occurrences of x are replaced by `x.arg`. We cannot directly access `arg` on the right-hand side of `fun`: this wouldn't be "contractive", because `arg` might not be initialized before `fun`. Hence, we pack t' in another class to ensure contractiveness. Now, an application $g(e)$ gets decomposed into three subsequent steps:

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

First we instantiate function `g` with a concrete argument yielding a thunk g_{app}. Then we evaluate this thunk by creating an instance g_{eval} of it. Finally we extract the result by querying field `res` of g_{eval}.

This example illustrates the correspondence between classes, methods and functors claimed by νObj.

III. Research Proposal

In our own research, we have been exploring a new core calculus for path-dependent types, the Dependent Object Types (DOT) calculus, which could serve as a new type-theoretic foundation for Scala and languages like it. The three calculi presented demonstrate that path-dependent types arise in diverse settings, and thus, it is valuable to identify a core calculus which captures their essence. In DOT, we are not concerned with modeling orthogonal concepts such as inheritance and mixin composition.

DOT aims to bring more uniformity and simplicity to Scala. Scala, and νObj which models it, 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). At first glance, this offers an economy of concepts. However, it is problematic because mixin composition and intersection types have quite different properties. In particular, we have seen in section II-C that mixin composition is not commutative, unlike classical intersection types.

In DOT, we replace Scala's compound types by classical intersection types, which are commutative. We also complement this by classical union types. Intersections and unions form a lattice wrt subtyping. This addresses another problematic feature of Scala: In Scala's current type system, least upper bounds and greatest lower bounds do not always exist. Here is an example: Given two traits

```
trait A { type T <: A }
trait B { type T <: B }
```

The greatest lower bound of `A` and `B` is approximated by the infinite sequence

```
A with B { type T <: A with B { type T <: A with B {
  type T < ...
}}}
```

The limit of this sequence does not exist as a type in Scala. This is problematic because glbs and lubs play a central role in Scala's type inference. The absence of universal glbs and lubs makes type inference more brittle and more unpredictable.

Our immediate goal is to prove type soundness of the DOT calculus. We first tried to prove the calculus type-safe using the standard theorems of preservation and progress [10], [7]. Unfortunately, for the DOT calculus and any variants that we devised, preservation doesn't hold. Indeed, we found many counterexamples to preservation due to narrowing, i.e. a type becoming more precise after substitution. However, the standard theorems of preservation and progress are just one way to prove type soundness. We are exploring a promising avenue to prove type safety using the powerful method of step-indexed logical relations [11], [12], [13]. In the process of proving the calculus sound, we are also refining it.

In future work, we plan to investigate translations of Scala features into the core DOT calculus, and implement an experimental compiler front-end which uses the DOT calculus for type-checking.

References

[1] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger, "An Overview of the Scala Programming Language (2. edition)," EPFL, Tech. Rep., 2006.

[2] M. Odersky, V. Cremet, C. Röckl, and M. Zenger, "A nominal theory of objects with dependent types," in *ECOOP*, 2003, pp. 201–224.

[3] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky, "A core calculus for scala type checking," in *MFCS*, 2006, pp. 1–23.

[4] M. Odersky and M. Zenger, "Scalable Component Abstractions," in *Proceedings of OOPSLA 2005*, 2005.

[5] R. Harper and M. Lillibridge, "A type-theoretic approach to higher-order modules with sharing," in *POPL*, 1994, pp. 123–137.

[6] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad, "Tribe: a simple virtual class calculus," in *AOSD*, 2007, pp. 121–134.

[7] B. C. Pierce, *Types and programming languages*. MIT Press, 2002.

[8] M. Lillibridge, "Translucent sums: A foundation for higher-order module systems," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, December 1996.

[9] N. R. Cameron, J. Noble, and T. Wrigstad, "Tribal ownership," in *OOPSLA*, 2010, pp. 618–633.

[10] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Inf. Comput.*, vol. 115, no. 1, pp. 38–94, 1994.

[11] A. J. Ahmed, "Semantics of types for mutable state," Ph.D. dissertation, Princeton University, 2004.

[12] ——, "Step-indexed syntactic logical relations for recursive and quantified types," in *ESOP*, 2006, pp. 69–83.

[13] C. Hritcu and J. Schwinghammer, "A step-indexed semantics of imperative objects," *Logical Methods in Computer Science*, vol. 5, no. 4, 2009.