Functional and Object-Oriented Approaches to Compositional Programming

Martin Odersky

Core Computer Science Institute
Ecole Polytechnique Federal de Lausanne

Contents

1. What is Compositional Programming?
2. Why is it important? What are the new challenges in programming?
3. A concrete language design: Scala.
4. Type systems for functional and object-oriented languages.
5. Duality: Parameterization vs. Abstract Members
6. Generics, virtual types, and family polymorphism.
7. A calculus for objects with dependent types.

Compositional Programming

As programs and systems grow larger, the techniques and principles of composition become more important than the individual parts.

Composition "in the small" is supported well by functional languages.

Witnesses:

Backus' 1977 Turing award lecture.
John Hughes: "Why functional programming matters."

On the other hand, component architectures for "assembly in the large" are typically object-oriented.

I argue that to make progress on composition, we should try to unify both approaches.

Hence, we should move from functional or object-oriented programming to compositional programming.

The Object-Oriented Approach

- Programs are composed from objects.
- Objects have members, e.g. named object references and methods.
- Objects encapsulate state, access is through methods.
- Objects are constructed from classes.
- Object interfaces are typed collections of methods.
- Composition principles:
  - aggregation (i.e. object A contains or refers to object B).
  - inheritance (i.e. class A inherits and potentially modifies behavior of class B)
  - recursion (object references may be recursive).
- Today, this is the standard approach, so much so that "object-oriented" becomes increasingly meaningless; instead of "object-oriented programming" one can simply say "programming".

Strengths and Limitations of the OO Approach

- Object-orientation is clearly suitable for systems modelling
  - Identify components, and model them with objects.
  - Identity component services and model them with methods, or, if more complex, with interfaces.
  - Widely adopted design methodology: UML.
- Today's component frameworks are object-oriented, e.g.
  - Corba
  - COM
  - Enterprise Java Beans
  - .NET
- Composition principles supported through design patterns, e.g.
  - Visitor, Publish-Subscribe, Factory, Wrapper, ...
- Common problem: Weak typing and weak specification of interfaces.

The Functional Approach

- Separation between (immutable) data and functions operating on data.
- Data are described by algebraic types.
- Functions operate by pattern matching.
- Functions can be higher-order; more complex functions can be constructed from simple ones using combinators.

Strengths and Limitations of the Functional Approach

- Flexible form of composition.
- Rich set of laws for program verification and transformation.
- Since components are functions, their interfaces can be accurately typed.
- Limitations:
  - Dealing with mutable state and concurrency requires additions to the basic theory.
  - Functions are inherently small components. How are they assembled into bigger ones?

    ➔ Need module systems, which leads to a stratification into core and kernel languages.

    ➔Often, the module language is to weak for flexible composition.

Contents

1. What is Compositional Programming?
2. Why is it important? What are the new challenges in programming?
3. A concrete language design: Scala.
4. Type systems for functional and object-oriented languages.
5. Duality: Parameterization vs. Abstract Members
6. Generics, virtual types, and family polymorphism.
7. A calculus for objects with dependent types.

Background: Global Computing

- Claim: *Web-based applications* represent a major paradigm shift in computing.
  - 1960's: *central* computing
  - 1980's: *local* computing
  - 2K's: *global* computing
- Global computing:
  - Computation is distributed between different sites in the internet.
  - Goal is cooperative behavior under a wide range of conditions.
- Examples:
  - internet auctions (e.g. Ebay),
  - travel reservation (e.g. Expedia),
  - making use of compute cycles (GRID),
  - group collaboration (e.g. Groove)
  - peer-to-peer information sharing.

New Challenges

Internet-wide distributed computing poses a series of challenges that are difficult to meet.

In particular,

- Participants in a computation (sites) may fail.
- Sites may join or leave the system at unpredictable times (e.g. P2P)
- Message delays may be unbounded.
- Failure detection is approximate.
- Transactional semantics needs to be maintained.

Our only weapons: immutability, larger granularity.

What does it mean for programming languages?

Looking back...

The last shift from central to local computing triggered a change in programming language paradigms from *structured* to *object-oriented* programming.

- Issue at the time:

  Extensible frameworks for graphical user interfaces require dynamic binding.

  Example: Window-manager (fixed) calls window's display method (variable)

- This form of extensibility is essential in

  Simulation (hence, Simula, 1967)

  Graphic User Interfaces (hence, Smalltalk, 1972-80)

... and the rest is history.

Recent Developments

Global computing has already influenced programming languages in important ways.

Driving force: mobile code must be portable and verifiably secure.

Hence, the need for

- strong type systems,
- complete runtime checking
- garbage collection
- precise language specifications
- verifiable implementations.

This has led to new mainstream languages: Java, and lately, C#.

In the future, increased need to develop foundations to avoid arbitrary and complicated language designs.

But will this be all?

Role of a Program

Claim: the current shift from local to global computing is also likely to change programming paradigms.

- New issues:
  Asynchronous communication, sharing information between sites, processing XML documents.

In the local computing setting

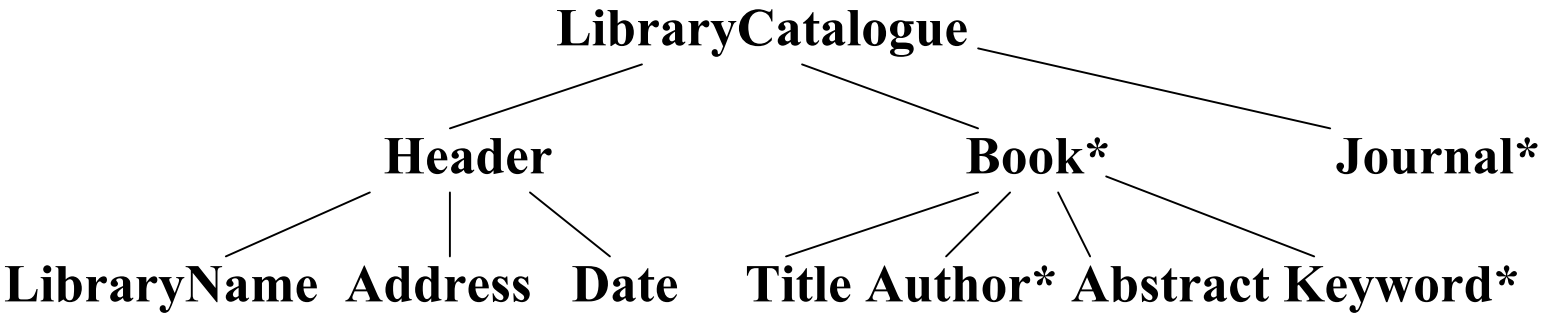
- Programs are objects.
- Communication is method invocation.
- Parameters are simple values or object references.

In the global computing setting

- Programs are still objects, but:
- Communication is by asynchronous message sends.
- Parameters are immutable structured documents (e.g. written in XML).
- Object references are problematic.

Example: Representing XML Documents

On an abstract level, an XML documents is simply a tree.



- We can expect standard software to convert between external documents and trees.
- Trees are *pure data*; no methods are attached to nodes.
- Trees should reflect the application domain, rather than being a generic "one size fits all" such as DOM.
- This means we want different kind of data types for different kinds of tree nodes.

Traversing XML Trees – The Standard Way

To return all books whose author's name is "Knuth":

```
if (node instanceof LibraryCatalogue) {
   LibraryCatalogue catalogue = (LibraryCatalogue) node;
   for (int i = 0; i < catalogue.elems.length; i++) {
      Node node1 = catalogue.elems[i];
      if (node1 instanceof Book) {
         Book book = (Book)node1;
         for (int j = 0; j < book.authors.length; j++) {
            if (book.authors[i] .equals("Knuth")) {
               result.append(book.title);
            }}}}
} else error("not a library catalogue");
return result.toList();
```

- Lots of type tests and type casts - ugly and inefficient.
- Alternative: Visitors, but these are heavyweight and hard to extend.

A Better Way to Traverse XML Trees

Declarative Programming: Specify what to compute, not how to do it.

```
node match {
  case LibraryCatalogue(header, books) =>
    for (b ← books, b.authors == “Knuth”) yield b.title
  case _ =>
    error(“not a library catalogue”)
}
```

Elements:

- pattern matching,
- recursion,
- higher-order functions

These are all known from functional programming, but they need to be generalized to an open world.

The State of the Art

- Current Practice: Use patchwork of different languages for web applications. E.g..

  C++ for the back-office,
  Java or C# for transactions and communication infrastructure,
  XQuery for database queries,
  XSLT for XML document transformations.

- Lots of little (or not so little) languages, each reasonably good at their job.

- But: bad things happen at cross-language interfaces.
  - Mismatches in operational semantics.
  - No static type system to control cross-language interfaces.

A Case for Multi-Paradigm Languages?

Instead of many different languages maybe one should use a multi-paradigm language?

Examples:

- Ada 95 (Imperative + OO)
- OCaml (Functional + OO)
- CLOS Common Lisp (Functional + OO)
- Oz (Logic + Functional + OO)
- Pizza (OO + Functional)
- ...

Problems:

- Size of resulting language.
- Semantic gaps at cross paradigm interfaces.

Scalable, Compositional Languages

- Goal: develop scalable languages which can grow with the needs of their users.
  - Simple example: Add a type for complex numbers.
  - More complicated examples:

    database querying, exception handling,

    process communication and monitoring.

- Observation: Since the language has to work "in the large" as well as "in the small", basic elements become unimportant. All that matters is the glue.

- Hence, scalable programming languages are inherently compositional.

- Thesis: scalable languages can be developed from a synthesis of object-oriented and functional programming.

Foundations for Compositional Languages

Recently, progress has been made on several fronts to obtain sound foundations for a synthesis between functional and object-oriented programming.

- Operational calculi:

  Join calculus, localized pi.
- Models of classes and objects

  (e.g. FOOL workshops)
- Type systems:

  e.g. Objective CAML, FGJ, gbeta, ...

Type Systems – Overcoming the Culture Clash

- Typical functional languages have
  - structural type systems,
  - powerful type inference methods a la Hindley/Milner,
  - powerful abstraction capabilities through abstract types
- Typical object-oriented languages have
  - nominal type systems,
  - no type inference,
  - powerful composition capabilities through first-class objects and unrestricted recursion.
- Challenge: Combine advantages of both in one system.
  - Formalization of nominal types.
  - Local type inference.
- Both were pioneered in the context of GJ, need to be extended now, in particular to abstract types.
- Candidate type system: ***νObj***, a nominal theory of objects.
- More on that on Wednesday.

Contents

1. What is Compositional Programming?
2. Why is it important? What are the new challenges in programming?
3. A concrete language: Scala.
4. Type systems for functional and object-oriented languages.
5. Duality: Parameterization vs. Abstract Members
6. Generics, virtual types, and family polymorphism.
7. A calculus for objects with dependent types.

Language Design for Compositional Programming

- Claim: Scalabaility stems from a unification (rather than addition) of features.
- At EPFL, we develop a new programming language with scalability as foremost design goal (Scala = "Scalable language").
- Scala is a statically typed, object oriented language which adapts many elements found in functional languages.
- Main elements:
  - (Mixin-)classes,
  - first-class functions,
  - first-class extensible pattern matching.
- Seamless interoperability with Java and .NET is a hard requirement if the language is supposed to be useful.

Scala by Examples: Classes

- Here is a simple class that defines rational numbers.

```
class Rational(n: Int, d: Int) with {
  private def gcd(x: Int, y: Int) = {
    if y < x then gcd(x % y, y)
    else if x < y then gcd(y % x, x)
    else x
  }
  private val g = gcd(n, d);
  val numer: Int = n / g;
  val denom: Int = d / g;
  def + (that: Rational) =
    new Rational(numer * that.denom + that.numer * denom,
                 denom * that.denom);
  def – (that: Rational) = ...
  ...
}
```

Notes

- Operators are methods.
- General operator syntax:

  a op b is equivalent to a.op(b)

- Examples:

  a + b

  msg append “.”

- One constructor per class, constructor parameters follow class name.
- Methods can be parameterless.

A Refinement

```
abstract class Ord[a] with {
   def < (that: a): Boolean;
   def <= (that: a) = this < that || this == that;
   def > (that: a) = that < this;
   def >= (that: a) = that <= this;
}
final class OrderedRational(n: Int, d: Int)
                extends Rational(n, d) with Ord[OrderedRational] with {
   override def == (that: Any) = |
      that.is[OrderedRational] &&
      numer == that.as[OrderedRational].numer &&
      denom == that.as[OrderedRational].denom;
   def < (that: OrderedRational) =
      numer * that.denom < that.numer * denom;
}
```

- **a** is type parameter representing type of object itself.
- **==** overrides a method in Object:

```
class Object with {
   def == (that: Any): Boolean = this eq that; ...
```

Mixin Composition

- OrderedRational is derived by mixin composition from Rational and Ord.
- Every class can be used as a mixin.
- When composed, its superclass is overwritten (covariantly).
- Therefore, in the mixin

  A with B

  A must be a subtype of B's declared superclass.
- Mixin composition provides essentially all of the benefits of multiple inheritance without its worst shortcomings (name clashes, class initialization).

Scala by Examples: Auction Service

- As a simplified example of a web-service application, consider a class of auction objects.
- Auctions receive and send messages from clients (sellers and bidders)
- Need to coordinate bids, inform clients.
- Messages are XML documents, converted internally to application-specific trees.
- Messages are represented by classes like these:

```
abstract class Message;
case class Offer(bid: Int, self: Actor) extends Message;
case class BestOffer extends Message;
...
```

A Class for Auctions

```
class Auction(seller: Actor, minBid: Int, closing: Date) extends Actor with
  def run = {
    var maxBid: Int = minBid – increment; var maxBidder: Actor = null;
    while (True) {
      receiveWithin((closing – Date.currentDate).msec) {
        case Offer(bid, client) =>
          if (bid >= maxBid + increment) {
            if (maxBidder != null) maxBidder send BeatenOffer(bid);
            maxBid = bid ; maxBidder = client;
            client send BestOffer;
          } else client send BeatenOffer(maxBid);
        case TIMEOUT =>
          if (maxBidder != null) {
            val reply = Sale(seller, maxBidder);
            maxBidder send reply; seller send reply;
          } else seller send NoSale;
    }}}}
```

Reserved words in **bold**; everything else is defined by user or libraries.

Abstraction Building

- This looks somewhat like a specialized language for web applications (with process communication operations in the style of Erlang).
  - Structured messages.
  - Flexible syntax for sending and receiving messages.
  - Timeouts are built in.
- But the keywords are not where one expects them to be!
- In fact, almost all abstractions used in the example are library abstractions, so they can be defined and adapted by users.
- Idea: Provide flexible schemes for abstraction and composition rather than the primitives themselves.

Bridging the Gaps

We now see three examples of where unification of constructs gives interesting results:

- Data Variation
- Functions and Queries
- Functions and Objects

Bridging the Gaps (1): Data Variation

- Functional languages express variation using algebraic data types and pattern matching.

```
data  Message = Offer(Int, Actor) | Inquire(actor) | ...
case (message) of
    Offer(x, client) => ...
    Inquire(client) =>
```

- Advantage: It is easy to define new *processors*.

- Object-oriented languages express variation using class hierarchies.

```
abstract class Message
class Offer extends Message { ... }
class Inquire extends Message { ... }
```

- Advantage: It is easy to define new *variants*.

Case Classes

- Like other object-oriented languages, Scala supports variation through class hierarchies.

- But it allows to *reverse* the construction process through case classes and pattern matching.

```
class Message;
case class Offer(bid: Int, client: Actor) extends Message;
case class Inquire(client: Actor) extends Message;

...

message match {
    case Offer(b, c) =>
    case Inquire(c) =>
}
```

- We thus get extensibility in both dimensions: it is easy to add new variants *and* new processors (this is essential for handling XML trees).

Pattern Matching

- Case classes can be decomposed using pattern matching.
- Example:

```
def sumLeaves(t: Tree) = t match {
  case Branch(l, r) => sumLeaves(l) + sumLeaves(r)
  case Leaf(x) => x
}
```

- **match** is a predefined method in Scala's root class Any defined as reverse function application:

```
class Any with { ...
  def match[a](f: (this)a): a = f(this)
```

- The { **case** ... } block defines a function of type (Tree)Int.

Implementation of Pattern Matching

- Since case classes can be defined anywhere in a program, we cannot associate them with fixed tags.
- So we cannot compile fixed jump tables for case expressions, in the way algebraic data types are compiled in functional languages.
- Instead, we assign tags to case classes at load time.
- Jump-tables also need to be adapted at load time.
- Possibilities:
  - Dispatch with binary search (requires sorting).
  - Dispatch with computed jump (requires computation of perfect hash functions on tags).

Bridging the Gaps (2): Functions and Queries

- Database queries are usually handled by special languages.
- Problem: Difficulty in interfacing when the query is made by a program instead of a user.
- Here's a query in Scala:

```
for (book ← books;
      book.title contains "XML";
      "Knuth" ← book.authors) yield book
```

- Queries like this are mapped by the Scala compiler into applications of three higher-order functions: map, flatMap and filter.

- For instance, here's the translation of the previous query:

```
books
  .filter (book => book.title contains "XML")
  .flatMap (book =>
    book.authors
      .filter (a => a == "Ullmann")
      .map (a => book))
```

- The mapping relies only on the existence of functions map, flatMap and filter as methods of the "queried" type.

⇒ The for-construct can equally well applied in other contexts, such as combinatorial search, matrix algebra, etc.

One only needs to define map, flatMap and filter for the carrier type.

Bridging the Gaps(3): Functions and Objects

- Scala is a pure object-oriented language: Every value is an object.
- Scala is a functional language: Functions are "first-class" values.
- Hence, functions in Scala are objects.
- Indeed, they are instances of class

```
abstract class Function1[a,b] with {
   def apply(x: a): b
}
```

- But what then is the status of the apply method in class Function?
- Methods are *not* functions. But a method is implicitly converted to a function when it is not immediately applied to its arguments.

Specialized Functions

- Since functions are non-final classes, they can be specialized.
- Example 1:

```
class Array[a](len: Int) extends Function[Int, a] with {
  def apply: a
  def update(i: Int, x: a)
  def length: Int
}
```

- Usage:

```
a(i) = a(i-1) + a(i+1)
```

- The compiler translates this into calls of **apply** and **update**.
- Analogous for hashtables, association lists, etc.

Partial Functions

- Another useful specialization of the function type is to partial functions.

- Partial functions have an isDefinedFor method, which is used to test whether the function is defined for a given argument.

```
abstract class PartialFunction[a,b] extends Function[a,b] with {
   def apply(x: a): b
   def isDefinedFor(x: a): Boolean
}
```

- Pattern matching case expressions are mapped by the compiler to partial functions when the context requires it.

- E.g. { **case** Offer(b, c) => b
  case Inquire(c) => -1 } is of type PartialFunction[Message, Int].

Application: Exception Handling

- Partial functions allow the definition of try-catch in the library.
- Example usage:

```
try {
   val data = f.read()
   process(data)
} except {
   case ex: IOException => out.println(ex.getMessage)
   case ex: ClassCastException => throw(new InternalError)
}
```

- The typed variable pattern ex: IOException matches any subtype of IOException and binds ex to it.

Implementing Try-Catch

- Here's an implementation of try-catch in terms of the try-catch of the underlying host language (written in ***italics***).

```
abstract class Except[a] with {
   def except(k: PartialFunction1[Exception,a]): a
   }
def try [a] (def block: a): ExceptFinally[a] = {
   try {
      val x = block
      new Except[a] with { def except(k) = x }
   } catch (Exception ex) {
      new Except[a] with {
         def except(k) = if (k isDefinedFor ex) k(ex) else throw(ex)
   }
}
```

- **def**-parameters indicate call-by-name evaluation of corresponding argument.

Application: Process Communication

- Message spaces are a high-level communication primitive.
- Operations:
  - send a message,
  - retrieve a message matching a pattern.
- Signature:

```
case class TIMEOUT extends Message;
class MessageSpace with {
   def send(msg: Any): Unit;
   def receive[a](f: PartialFunction[Any, a]): a;
   def receiveWithin[a](msec: Long)(f: PartialFunction[Any, a]): a;
}
```

Erlang-Style Processes

- An Actor is simply obtained by a mixin composition of a thread and a message space.

```
abstract class Actor extends Thread with MessageSpace;
```

- This gives a process model in the style of Erlang.
- Properties:
  - Scalable to many threads, many message kinds, structured messages.
  - Processes have only a single channel for incoming messages, hence it is easy to compose processes for forwarding, filtering, logging, etc.
- But where Erlang's process model is fixed, Scala's is a library abstraction.
  - Required: (mixin-)classes, (partial-)functions, pattern matching.

Auctions Revisited

The auction class is now revealed as an application of the actor abstract

```
class Auction(seller: Actor, minBid: Int, closing: Date) extends Actor with
  def run = {
    var maxBid: Int = minBid – increment; var maxBidder: Actor = null;
    while (True) {
      receiveWithin((closing – Date.currentDate).msec) {
        case Offer(bid, client) =>
          if (bid >= maxBid + increment) {
            if (maxBidder != null) maxBidder send BeatenOffer(bid);
            maxBid = bid ; maxBidder = client;
            client send BestOffer;
          } else client send BeatenOffer(maxBid);
        case TIMEOUT =>
          if (maxBidder != null) {
            val reply = Sale(seller, maxBidder);
            maxBidder send reply; seller send reply;
          } else seller send NoSale;
    }}}}
```

History

- Scala evolved from our work on functional-nets
- First goal was to build a minimal programming language based on a universal calculus of computation.
- We relied on known encodings to obtain pattern matching and classes.
- User experience taught us that encodings work fine, but only for experts.
- Scala tries to have wider appeal by including more constructs in the base language.
- New question: How to combine object-oriented and functional constructs and how to unify them where possible.
- This has proved to be much more interesting than expected!

Conclusion

- Compositional languages push the enevlope in what new abstractions can be built by library designers.
- They make it possible to express many domain specific languages as libraries.
- Of course, this expressiveness needs to be used intelligently and responsibly – it is just as easy to build a bad library than it is to build a bad language.
- But, given the choice, I prefer a bad library because it is easier changed or replaced.

Scala Project Status

- Compiler for a language subset of Scala exists (you'll use it in the lab).
- We also used it in mandatory course "Programmation IV" for 2nd year students at EPFL.
- Currently, the compiler produces its own intermediate language, which is (slowly) interpreted.
- Compilers for Java bytecodes and .NET are under development.
- Compiler maps as directly as possible to underlying architecture.

⇒ Easy interoperability by cross-language method invocation and inheritance.

- To find out more: lampwww.epfl.ch/scala

Summer School on Generic Programming

Functional and Object-Oriented Approaches to Compositional Programming

Part II: Parameterization vs Named Abstraction

Martin Odersky
EPFL

Parameterization

Parameterized functions and their applications are the central concept in functional programming.

For instance, here are the only two reduction rules of system F, the theory underlying typed functional languages.

$$\begin{array}{lcl} (\lambda x : T.e)\ d & \rightarrow & [d/x]\ e \\ (\Lambda X.e)[T] & \rightarrow & [T/X]\ e \end{array}$$

Or, in Scala:

```
def inc(value: Int) = value + 1 ;
inc(2)
```

Parameterization relies on positional abstraction and instantiation.

Named Abstraction

In object oriented languages, one can use alternatively named abstraction and implementation.

```
abstract class Incrementer with {
  def value(): Int;
  def result = value() + 1;
}
new Incrementer with { def value() = 2 }.result
```

This is called an anonymous class in Java (with syntax very similar to the above).

In other languages one needs to use a named subclass; e.g.

```
class MyIncrementer extends Incrementer with {
  def value(): Int;
  def result = value() + 1;
}
new MyIncrementer;
```

Generalizing Named Abstraction

In mainstream OO-languages such as Java, there is a strict separation of capabilities.

- Parameterization only for basic values, objects, and (in GJ) types.
- Named abstractions only for methods.

But there is no inherent reason why this should be so!

For instance, it should be possible to abstract over values:

```
abstract class Incrementer with {
    val value: Int;
    def result = value + 1;
}
new Incrementer with { value = 2 }.result
```

Named Abstraction over Types

Further, it should also be possible to abstract over types:

```
abstract class MutableSet {
    type elem;
    def incl(x: elem): Unit;
    def contains(x: elem);
}
val s = new MutableSet with { type elem = Int }
s incl 2; s incl 37;
if (s contains 1) ...
```

What does this remind you of?

Strengths of Parameterization

Here is something which is natural using parameterization and a bit more convoluted using named abstraction.

We change our *MutableSet* class to a *Set* class where *incl* returns a new set instead of changing the existing one.

Here's the program using a type parameter:

```
abstract class Set[elem] {
  def incl(x: elem): Set[elem];
  def contains(x: elem);
}
val s = new Set[Int]
if (s.incl(2).incl(37).contains(1)) ...
```

Here's the program using named abstraction:

```
abstract class Set {
    type elem;
    def incl(x: elem): Set with { type elem = outer.elem };
    def contains(x: elem);
}
val s = new Set with { type elem = Int }
if (s.incl(2).incl(37).contains(1)) ...
```

Notes:

- The term { ***type*** *elem* = ***outer***.*elem* } denotes a record type, with a single type field, *elem*.
- It is in form and function quite similar to a sharing constraint in SML.
- ***outer*** refers to the value of ***this***, as it is seen in the class block directly enclosing the current one.
- Types are members of values; hence ***outer***.*elem* is a legal type.
- We will need a system of dependent types to make sense of this.

Mapping Between Positional and Named Abstraction

Generally, we can map every program with parameterization into one with named abstraction by a purely local transformation, at a modest increase in size.

What about the other direction?

Claudio Russo has shown in his thesis that SML dependent types can be translated to a program with parameterization.

But his translation is not local — translating a named abstraction in the middle of a program might lead to changes in unrelated places of the program.

Strengths of Named Abstraction

Here's something that is not that hard to do using named abstraction and very difficult to do using parameterization.

Task: Construct an abstraction for lists with alternating elements of types X and Y.

Every list node which contains an X element is followed by ***null*** or a node which contains a Y element, and *vice-versa*.

Then, extend the pair of classes to also include a *length* function.

The following attempt shows why this is not trivial.

First Attempt

Let's arbitrarily choose some types X and Y.

```
type X = Int;
type Y = String;
```

A natural pair of classes for alternating lists would be:

```
class XList(h: X) with {
  def head: X = h;
  var tl: YList = null;
  def tail: YList = tl;
  def setTail(t: YList) = {tl = t}
}
```

```
class YList(h: Y) with {
  def head: Y = h;
  var tl: XList = null;
  def tail: XList = tl;
  def setTail(t: XList) = {tl = t}
}
```

So far, everything is fine.

But now, try to extend the pair of classes:

```
class XListWithLen(h: X) extends XList(h) with {
  def len: Int = (if (tail == null) 0 else tail.len) + 1
}
```

(and analogously for *YList*).

Unfortunately, the Scala compiler responds:

```
xylist00.scala:19: len is not a member of YList
  def len: Int = (if (tail == null) 0 else tail.len) + 1
                                               ^
```

Avoiding the Problem

- The standard OOP way to address the problem is to replace static with dynamic typing, by using a type cast at this point.

```
class XListWithLen(h: X) extends XList(h) with {
  def len: Int = (if (tail == null) 0 else
                  tail.as[XListWithLen].len) + 1
}
```

- Of course, this is an admission of defeat – either our program is written in an illogical way, or our static type system was not expressive enough to express what we want.

- The standard functional way is not to have inheritance at all. So *XList*, *YList* cannot be re-used in extensions.

A Solution Using Nested Types

To avoid the static type cast, we have to make *XList* generic in the type of its tail.

We know that the tail of an *XList* is some subtype of *YList*, but we do not know which one.

So, introduce an abstract type for it, which is contained in a new class *XY*.

```
abstract class XY with {
  type xlist extends XList;
  type ylist extends YList;
```

This uses bounded abstraction, where the type variable *xlist* can be instantiated with an arbitrary subtype of *XList*.

The bound classes are defined almost as before:

```
abstract class XY with {
  type xlist extends XList;
  type ylist extends YList;

  class XList(h: X) with {
    def head: X = h;
    var tl: ylist = null;
    def tail: ylist = tl;
    def setTail(t: ylist) = {tl = t}
  }
}
```

```
class YList(h: Y) with {
  def head: Y = h;
  var tl: xlist = null;
  def tail: xlist = tl;
  def setTail(t: xlist) = {tl = t}
}
```

Now, extend *XY* to a class where *XLists* and *YLists* have lengths.

```
abstract class XYWithLen extends XY with {
  type xlist extends XListWithLen;
  type ylist extends YListWithLen;

  class XListWithLen(h: X) extends XList(h) with {
    def len: Int = (if (tail == null) 0 else tail.len) + 1
  }
  class YListWithLen(h: Y) extends YList(h) with {
    def len: Int = (if (tail == null) 0 else tail.len) + 1
  }
}
```

The abstract type *xlist* in *XYWithLen* overrides *xlist* in *XY*.

Overriding requires that the bound in the subclass is $\leq$ than the bound in the superclass.

Because of their bounds, *xlist* and *ylist* are known within *XYListLen* to contain *len* fields.

So the selection *tail.len* is type correct.

Concrete Pairs of Alternating Lists

Before being able to create and use alternating lists, we still need to create a concrete value that contains the list classes.

In Scala, there are at least two ways of doing this.

- By creating a value:

  val *xy* = ***new*** *XYWithLen* ***with*** {
  type *xlist* = *XListWithLen*; ***type*** *ylist* = *YListWithLen*
  }

- By creating a module:

  module *xy* ***extends*** *XYWithLen* ***with*** {
  type *xlist* = *XListWithLen*; ***type*** *ylist* = *YListWithLen*
  }

The two expressions mean the same, with the exception that modules are created lazily, the first time one of their members is needed.

N.B. This is exactly what happens with the initialization of static class members in Java.

Using Alternating Lists

Now, we can use the alternating list abstraction:

```
val xs = new xy.XListWithLen(1);
cd.setTail(new xy.YListWithLen("a"));
l.len
```

The Scala interpreter responds with 2, as expected.

Type Safety

The static type systems ensures that *XListWithLens* can be followed only by *XListWithLens*, not by arbitrary *XLists*.

So, if we try:

```
def breakit = {
  val xsl = new xy.XListWithLen(2);
  val xs: XList = xsl;
  xs.setTail(new xy.YList("a"));
  xsl.len
}
```

we get:

```
xylist2.scala:39: type mismatch;
  found   : xy.YList
  required: xy.ylist
  xs.setTail(new xy.YList("a"));
             ^
```

Virtual Types and Family Polymorphism

The technique we have used here has been first developed in the context of BETA [*Madsen 83*].

However, in BETA one does not distinguish between the abstract type *xlist* and the bound class *XList*.

Instead, one overrides the nested class *XList* with a subclass.
(Such overridable classes are called virtual types in BETA).

As a consequence, the *breakit* example would be statically type correct in BETA, and would then lead to a run-time error.

A later design in GBETA [*Ernst, Thesis*] avoids the type hole by following essentially the rules proposed here.

The technique of collecting related classes in an outer classes, so that they can be specialized together is called family polymorphism
[*Ernst, ECOOP'01*].

Why Does This Matter?

The example of alternating lists might seem artificial; however, problems like this occur in many larger systems.

Example: Publish-Subscribe.

A publish-subscribe system consists of a publisher and an arbitrary number of subscribers.

Subscribers register themselves with a publisher.

A publisher originates events and notifies all subscribed entities of them.

Publish-subscribe is the basic communication mechanism of

- window Toolkits such as AWT, Swing, .NET Windows,
- Enterprise Java Beans,
- many distributed middleware systems.

Issues of Static Typing

Here's a statically typed framework for publish-subscribe:

```
abstract class PublishSubscribe with {
  type publisher extends Publisher;
  type subscriber extends Subscriber;

  class Publisher with {
    var vs: List[subscriber] = [];
    def registerSubscriber(v: subscriber) = vs = v :: vs;
  }

  abstract class Subscriber with {
    def update(m: publisher): Unit
  }

  def changed(m: publisher) =
    m.vs foreach (v ⇒ v.update(m));
}
```

Note that it's the same schema as for alternating lists:

- *Publisher* refers via its *registerSubscriber* method to the type of its subscribers.
- *Subscriber* refers via its *update* method to the type of its publishers.
- So both subscribers and publishers need to be specialized together.

Here's an example of a specialization:

```
class Draw extends PublishSubscribe with {
    type publisher = DrawPublisher;
    type subscriber = DrawSubscriber;
    class DrawPublisher extends Publisher with {
        def text = "xyz";
    }
    class DrawSubscriber extends Subscriber with {
        def update(m: publisher): Unit =
            System.out.println(m.text + " has changed");
    }
}
```

Another Example: Graphs

- A graph consists of nodes and edges.
- A node refers via operations such as *sucessors* to the type of edges connected to it.
- An edge refers via operations such as *from*, *to* to the type of nodes connected to it.
- So both classes should be specialized together.
- This will be the subject of the lab session.

Limitations of Parameterization

You might ask: "So why can't one do the same with traditional generic types and methods?".

It can be done, but it is much harder.

For instance, let's try for alternating lists:

- Instead of having abstract type members *xlist* and *ylist*, we have type parameters.

```
class XList[ylist extends YList]
                (h: X) with {
  def head: X = h;
  var tl: ylist = null;
  def tail: ylist = tl;
  def setTail(t: ylist) = {tl = t}
}
```

```
class YList[xlist extends XList]
                (h: Y) with {
  def head: Y = h;
  var tl: xlist = null;
  def tail: xlist = tl;
  def setTail(t: xlist) = {tl = t}
}
```

... and, to extend:

```
class XListWithLen[ylist extends YListWithLen]
                        (h: X) extends XList[ylist](h) with {
  def len: Int = (if (tail == null) 0 else tail.len) + 1
}

class YListWithLen[xlist extends XListWithLen]
                        (h: Y) extends YList[xlist, ylist](h) with {
  def len: Int = (if (tail == null) 0 else tail.len) + 1
}
```

Why does this not work?

... and, to extend:

```
class XListWithLen[ylist extends YListWithLen]
                        (h: X) extends XList[ylist](h) with {
  def len: Int = (if (tail == null) 0 else tail.len) + 1
}

class YListWithLen[xlist extends XListWithLen]
                        (h: Y) extends YList[xlist, ylist](h) with {
  def len: Int = (if (tail == null) 0 else tail.len) + 1
}
```

Here's what the Scala compiler has to say:

```
xylist02.scala:7: class XList takes type parameters.
class YList[xlist extends XList] (h: Y) with
                          ^

...
xylist02.scala:13: class YListWithLen takes type parameters.
class XListWithLen[ylist extends YListWithLen](h: X)
                                 ^
```

Second Attempt

The problem is that *XList* and *YList* are type constructors, not types.

So they cannot be used as bounds of type parameters.

We need to add proper parameters to the bounds *XList*, i.e.

XList [*ylist* ***extends*** *YList* [*xlist*]]
YList [*xlist* ***extends*** *XList* [*ylist*]]

This still does not work, since *xlist*, *ylist* are undefined in the bounds.

Third Attempt

We need to thread *xlist*, *ylist* through both classes:

$XList\,[xlist\ \textbf{extends}\ XList\,[xlist,\ ylist\,],\ ylist\ \textbf{extends}\ YList\,[xlist,ylist\,]\,]$
$YList\,[xlist\ \textbf{extends}\ XList\,[xlist,\ ylist\,],\ ylist\ \textbf{extends}\ YList\,[xlist,ylist\,]\,]$

Now, we can complete the definition of *Xlist* and *XListWithLen* as follows.

$\textbf{class}\ XList\,[xlist\ \textbf{extends}\ XList\,[xlist,\ ylist\,],$
$\qquad ylist\ \textbf{extends}\ YList\,[xlist,\ ylist\,]\,]\ \ (h : X)\ \textbf{with}\ \{$
"same as before"
$\}$
$\textbf{class}\ XListWithLen\,[xlist\ \textbf{extends}\ XListWithLen\,[xlist,\ ylist\,],$
$\qquad ylist\ \textbf{extends}\ YListWithLen\,[xlist,\ ylist\,]\,]\ \ (h : X)$
$\textbf{extends}\ XList\,[xlist,\ ylist\,]\,(h\,)\ \textbf{with}\ \{$
"same as before"
$\}$

and analogously for *YList* and *YListWithLen*.

This is still not enough; we also need to create leaf classes, which contain the proper instantiations of parameters.

```
class XListWithLenLeaf (h: X)
  extends XListWithLen[XListWithLenLeaf, YListWithLenLeaf](h) with {}
class YListWithLenLeaf (h: Y)
  extends YListWithLen[XListWithLenLeaf, YListWithLenLeaf](h) with {}
```

Finally, everything is in place for the test program.

```
val xys = new XListWithLenLeaf(1);
xys.setTail(new YListWithLenLeaf("a"));
xys.len;
```

Analysis

We have seen that family polymorphism can be emulated with generic types.

But there is a cost: Type signatures increase quadratically in size with the number of classes that need to be specialized together.

Why do named abstractions better?

Two essential differences:

- Nesting of types in classes
- Abstract classes are types instead of type constructors.

The first difference leads to a cleaner structuring.

But the quadratic blowup of types is avoided with named abstraction even if *XList* and *YList* are top-level.

Summer School on Generic Programming

Functional and Object-Oriented Approaches to Compositional Programming

Part III: Foundations for Objects with Abstract Types.

Martin Odersky
EPFL

Observation

Objects with abstract types model the essential capabilities of SML modules.

Unlike SML modules, they also support

- mixin-composition,
- recursive references within and between objects,
- first-class modules, since objects are values.

Goal of this part: Develop a type-systematic foundation for such objects.

The foundation is a dependent type system:

> If L is a type label and p is an object reference, then $p.L$ is a type which depends on p.

First step: Develop a language and a name-passing operational semantics for objects and classes.

νObj 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
	$[x : S \mid \overline{d}]$	Class template
	$t \,\&_S\, u$	Composition
d	$::=$	Definition
	$l = t$	Term definition
	$L \preceq T$	Type definition
p	$::=$	Path
	$x \mid p.l$	
v	$::=$	Value
	$x \mid [x : S \mid \overline{d}]$	

What's Missing

- Functions and function abstractions can be encoded using classes.
- Here's an idea how λ-calculus can be encoded.

$$
\begin{array}{lcl}
\langle\!\langle \lambda x : T.t \rangle\!\rangle & = & [x : \{arg : \langle\!\langle T \rangle\!\rangle\} \mid res = \langle\!\langle E \rangle\!\rangle] \\
\langle\!\langle t\ u \rangle\!\rangle & = & \nu x \leftarrow \langle\!\langle t \rangle\!\rangle \ \&\ [arg = \langle\!\langle u \rangle\!\rangle]\ ;\ x.res \\
x & = & x.arg
\end{array}
$$

  (in fact; this is not quite right; an additional indirection is needed to satisfy the contractiveness requirement for classes.)
- Parameterized types are encoded as types with abstract type members.
- Polymorphic functions are encoded as classes with abstract type members.
- In this way the whole of $F_{<:}$ can be encoded in νObj.

Operational Semantics of νObj

Structural Equivalence α-renaming of bound variables x, plus

$$\text{(extrude)} \qquad e\langle \nu x \leftarrow t \,;\, u\rangle \quad \equiv \quad \nu x \leftarrow t \,;\, e\langle u\rangle \qquad \textbf{if } x \notin fn(e),\, bn(e) \cap fn(x,t) = \emptyset$$

Reduction

$$\text{(select)} \qquad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \,;\, e\langle x.l\rangle \quad \rightarrow \quad \nu x \leftarrow [x : S \mid \overline{d}, l = v] \,;\, e\langle v\rangle \qquad \textbf{if } bn(e) \cap fn(x,v) = \emptyset$$

$$\text{(mix)} \qquad [x : S_1 \mid \overline{d}_1] \,\&_S\, [x : S_2 \mid \overline{d}_2] \quad \rightarrow \quad [x : S \mid \overline{d}_1 \uplus \overline{d}_2]$$

where evaluation context

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

Notes

- $\uplus$ is concatenation with overwriting of common labels:

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

- Side conditions on reduction rules ensure that free variables are not captured.
- Reduction $\longrightarrow$ is the smallest reflexive and transitive relation that satisfies rules (select) and (mix) and that is closed under formation of evaluation contexts:

$$t \longrightarrow u \qquad \text{implies} \qquad e\langle t \rangle \longrightarrow e\langle u \rangle$$

Theorem: $\longrightarrow$ is confluent:

If $t \twoheadrightarrow t_1$ and $t \twoheadrightarrow t_2$ then there exists a term t' such that $t_1 \twoheadrightarrow t'$ and $t_2 \twoheadrightarrow t'$.

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.
$L <: T$	The type label L represents an abstract type which is bounded by type T.

The right hand side of a $\prec$ or $<:$ binding can be recursive.

By contrast, recursive aliases are disallowed.

Example: Here's a simple type definition for lists of integers.

$$List \prec \{\ isEmpty\colon Boolean,\ head\colon Int,\ tail\colon List\ \}$$

L is the name of a nominal type.

Two aspects of $(\prec)$:

- L is a subtype of the record type $\{\ isEmpty\colon Boolean,\ head\colon X,\ tail\colon L\ \}$.
- Objects of type L can be created from classes that define fields *head* and *tail* with the given types.

Question: How does one create a subtype of nominal type?

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

First attempt:

$$\begin{aligned} \mathit{ListWithLen} \prec \{\ & \mathit{isEmpty} : \mathit{Boolean}, \\ & \mathit{head} : \mathit{Int}, \\ & \mathit{tail} : \mathit{ListWithLen}, \\ & \mathit{length} : \mathit{Int}\ \} \end{aligned}$$

In this case, *ListWithLen* is a subtype of *List*'s expansion,

$\{\ \mathit{isEmpty} : \mathit{Boolean},\ \mathit{head} : X,\ \mathit{tail} : L\ \}$.

But it is not a subtype of *ListWithLen* itself.

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

Example:

$$ListWithLen \prec List \;\&\; \{\; tail\colon ListWithLen,\; length\colon Int \;\}$$

is a subtype of $List$ as well as $\{\; tail\colon ListWithLen,\; length\colon Int \;\}$.

It has four fields:

$isEmpty\colon Boolean$ and $head\colon Int$, which come from $List$,
$tail\colon ListWithLen$,
$length\colon Int$.

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.

νObj 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
	$[x : S \mid \overline{d}]$	Class template
	$t\ \&_S\ u$	Composition
d	::=	Definition
	$l = t$	Term definition
	$L \preceq T$	Type definition
p	::=	Path
	$x \mid p.l$	
v	::=	Value
	$x \mid [x : S \mid \overline{d}]$	

L, M, N		Type label
S, T, U	::=	Type
	$p.\mathbf{type}$	Singleton
	$T \bullet L$	Type selection
	$\{x \mid \overline{D}\}$	Record type ($=:\ R$)
	$[x : S \mid \overline{D}]$	Class type
	$T\ \&\ U$	Compound type
D	::=	Declaration
	$l : T$	Term declaration
	$L \preceq: T$	Type declaration
$\preceq:$	::=	Type binder
	$=$	Type alias
	$\prec$	New type
	$<:$	Abstract type

Remarks

- The singleton type $p.L$ represents the set consisting of just the object referenced by path p.
- $p.L$ is syntactic sugar for $p.\mathbf{type}\bullet L$.
- $\{x \mid \overline{D}\}$ is an object type where the name x refers to the object itself (i.e. it corresponds to ***this*** or *self* in OO languages).
- References of one object declaration to another always go via self, i.e.

  $$\{\ x \mid L = \mathit{String};\ m = x.L\ \}$$

- $[x : S \mid \overline{D}]$ is a class type 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.

Typing Judgments

$\Gamma \vdash t : T$ Term t has type T in environment Γ.

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

Auxiliary Judgments

$\Gamma \vdash T\ wf$ Type T is well-formed.

$\Gamma \vdash T \ni D$ Type T contains declaration D.

$\Gamma \vdash T \prec U$ 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)

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

(VarPath)

$$\frac{\Gamma \vdash x : R}{\Gamma \vdash x : x.\mathbf{type}}$$

(SelPath)

$$\frac{\Gamma \vdash t : p.\mathbf{type}, \quad t.l : R}{\Gamma \vdash t.l : p.l.\mathbf{type}}$$

(Sub)

$$\frac{\Gamma \vdash t : T, \quad T \leq U}{\Gamma \vdash t : U}$$

(New)

$$\frac{\Gamma \vdash t : [x : S \mid \overline{D}], \quad S \prec \{x \mid \overline{D}\} \qquad \Gamma, x : S \vdash u : U \qquad x \notin \mathrm{fn}(U)}{\Gamma \vdash (\nu x \leftarrow t \,;\, u) : U}$$

(Class)

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

(&)

$$\frac{\Gamma \vdash t_i : [x : S_i \mid \overline{D}_i] \qquad \Gamma \vdash S \ wf, \quad S \leq S_i \qquad (i = 1, 2)}{\Gamma \vdash t_1 \,\&_S\, t_2 : [x : S \mid \overline{D}_1 \uplus \overline{D}_2]}$$

Path-Dependent Types

Question 1: Given $p : \{this \mid M <: \{\}, m : this.M\}$
what is the type of $p.m$?

Answer: $p.M$.

Question 2: Given $p : \{this \mid M = String, m : this.M\}$
what is the type of $p.m$?

Answer: $p.M$ or $String$ (they are the same).

Question 3: Given an expression $e = \nu x \leftarrow t \; ; \; x$,
of type $\{this \mid M = String, m : this.M\}$, what is the type of $e.m$?

Answer: $String$.

Question 4: Given an expression $e = \nu x \leftarrow t \; ; \; x$,
of type $\{this \mid M <: \{\}, m : this.M\}$, what is the type of $e.m$?

Answer: $e.m$ is not typable.

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-$\ni$)

$$\frac{\Gamma \vdash p.\mathbf{type} \leq \{x \mid \overline{D'}, D\}}{\Gamma \vdash p.\mathbf{type} \ni [p/x]D}$$

(Other-$\ni$)

$$\frac{\Gamma, x : T \vdash x.\mathbf{type} \ni D \qquad x \notin fn(\Gamma, D)}{\Gamma \vdash T \ni D}$$

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

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

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

Properties of νObj

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 $t \twoheadrightarrow a$, for some answer a such that $\epsilon \vdash a : T$.

Theorem: It is undecidable whether $\Gamma \vdash t : T$.

Proof by reduction to the problem in $F_{<:}$.

Summary

νObj is a nominal theory of objects with dependent types.

Nominal means two things:

- The operational semantics uses name passing instead of value passing.
- There is a type binder $\prec$, which creates nominal types.

The theory can express

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

The Exercises

We have prepared a small exercise to get to know Scala.

The get started, point your webbrowser to the file:

file://ecslab/demo67/exercise.html