12 Types

We can think of types as classifying values: every value belongs to a type describing the characteristics of the value and what operations can be performed with them. Primitive types generally include integers, floating point numbers, strings, Booleans; while more complex types correspond to structures combining values: tuples, lists, records, objects.

The main purpose of types is to ensure *type safety*: that operations are never applied to inappropriate values (e.g., concatenating integers as opposed to strings, or trying to access the first element of a Boolean). In object-oriented languages, not invoking a method on an object that does not implement that method is also generally considered part of type safety.

There are two ways to enforce type safety. A **dynamically-typed language** will check for a type error as late as possible, generally right before the problem manifests itself, throwing an exception (or using whatever langugage mechanism exists for reporting run-time errors). Thus, an error is reported right before a function is called with a value of an incorrect type. One advantage of this approach is flexbility: there is little in the way of writing code in such a language. You can write nonsense like `"hello" / true` and the language will not complain. At least, it will not complain until you run that part of the code. If that nonsense line is hidden away in a piece of code that does not execute very often, it may take a while for the error to appear — maybe never, in fact. This points to one disadvantage of this approach: it is very easy to write code that causes errors that are not caught during testing, if they occur on infrequent paths.

A **statically-typed language**, on the other hand, tries to check for type errors as soon as possible, before the code is even run. This is often a phase right before compilation. The system will analyze the code and try to determine that *none* of the code paths cause a type error. This yields very strong guarantees: no execution causes a type error.. But there is a cost. Because the problem of determining of a program has a type error anywhere is undecidable (it is a direct consequence of the undecidability of the halting problem, if you know about that), it is impossible to write a program analyzer that looks at the source code of a program and decides whether there is a type error anywhere. To get around this impossibility results, statically-typed languages implement a type-checking algorithm that sometimes gets it wrong — it will flag false positives. In other words, statically-typed languages may reject programs without type errors that it cannot prove have no type errors. An example is:

```
if (f(10) == 0) {
   return "hello";
} else {
   return "hello" / true;
}
```

If `f(10)` always yields value 0, then the problematic expression `"hello" / true` can never be executed. Most languages with a static type system will reject the above code, even if it is the case that `f(10)` is in fact always 0.

To drive the type checking algorithm, statically-typed languages often require type annotations to be supplied in the code, usually at binding sites such as function declarations.

The languages we've been developing until now have been dynamically typed. You can find the types in the values themselves. Our `Value` class has methods such as `isInteger()` to determine the type of any given value. These types are checked before every operation.

We now implement a static type system for our FUNC language. Because a static type system determines whether there are type errors before execution, types cannot be associated with values. Instead, we will associate types with *expressions*, and intuitively, the type of an expression will be the type of the values that would result from evaluating the expression. Static type systems are implemented using two related functions, which we take as methods on the expression class:

- `typeCheck(t)` — which checks that an expression does not have type errors and has type t

- `typeOf()` — which checks that an expression does not have type errors and returns its type

If we have an implementation of `typeOf`, we can derive an implementation of `typeCheck` pretty easily: to check if an expression has type `T`, get its type with `typeOf` and check if the obtained type is equal to `T`. Some languages and type systems are such that `typeOf` cannot always return a type, and in those cases `typeCheck` needs to be implemented directly. Our type system is simple enough that `typeOf` will be implementable and will induce the obvious `typeCheck`.

Here is our revised Exp class with the new methods;

```
abstract class Exp {

   def eval (env : Env[Value]) : Value

   def error (msg : String) : Nothing = {
     throw new Exception("Eval error: "+ msg + "\n   in expression " + this)
   }

   def terror (msg : String) : Nothing = {
     throw new Exception("Type error: "+ msg + "\n   in expression " + this)
   }

   def typeOf (symt:Env[Type]) : Type

   def typeCheck (t:Type, symt:Env[Type]) : Boolean = {
      val t2 = this.typeOf(symt)
      return t.isSame(t2)
   }
}
```

We will explain the additional parameter symt to typeOf() and typeCheck() below.

Since typeOf() returns a type, we need a representation for those types. We take the usual approach of defining an abstract base class Type whose subclasses implement the various kinds of types. We focus on integers, Booleans, integer vectors, and functions.

```
abstract class Type {

  def isSame (t:Type) : Boolean

  def isInteger () : Boolean = return false
  def isBoolean () : Boolean = return false
  def isIntVector () : Boolean = return false
  def isFunction () : Boolean = return false

  def funParams () : List[Type] = {
    throw new Exception("Type error: type is not a function\n   "+this)
  }

  def funResult () : Type = {
    throw new Exception("Type error: type is not a function\n   "+this)
  }
}


object TInteger extends Type {

  override def toString () : String = "int"

  def isSame (t:Type):Boolean = return t.isInteger()
  override def isInteger () : Boolean = true
}


object TBoolean extends Type {

  override def toString () : String = "bool"

  def isSame (t:Type):Boolean = return t.isBoolean()
  override def isBoolean () : Boolean = true
}


object TIntVector extends Type {

  override def toString () : String = "ivector"

  def isSame (t:Type):Boolean = return t.isIntVector()
  override def isIntVector () : Boolean = true
}


class TFunction (val params:List[Type], val result:Type) extends Type {

  override def toString () : String =
   "(fun "+params.addString(new StringBuilder(),"("," ",")").toString() + " " +
    result + ")"

  def isSame (t:Type):Boolean = {

   if (!t.isFunction()) {
     return false
   }

   if (t.funParams().length != params.length) {
     return false
    }
    for ((t1,t2) <- t.funParams().zip(params)) {
      if (!t1.isSame(t2)) {
        return false
      }
    }
    return t.funResult().isSame(result)
  }

  override def isFunction () : Boolean = return true
  override def funParams () : List[Type] = return params
  override def funResult () : Type = return result
}
```

This is all pretty straightforward. Note that we have predicates to query the type, and an `isSame()` method to implement type equality. A function type describes the types of the parameters to the function and the return type of the function. We have two methods to extract the type parameters and result types out of such a function type.

Here is the implementation of `typeOf()` for every expression we have. Literals are easy:

```
class EInteger (val i:Integer) extends Exp {
   ...

   def typeOf (symt:Env[Type]) : Type =
      TInteger
}


class EBoolean (val b:Boolean) extends Exp {
   ...

   def typeOf (symt:Env[Type]) : Type =
      TBoolean
}
```

A vector is a step more complex. Since we only have integer vectors, we first check that every expression used to build the vector is an integer, before returning the type of the vector expression to be an integer vector:

```
class EVector (val es: List[Exp]) extends Exp {
   ...
```

```
  def typeOf (symt:Env[Type]) : Type = {
    for (e <- es) {
      if (!e.typeCheck(TInteger,symt)) {
        terror("Vector component not an integer")
      }
    }
    return TIntVector
  }
}
```

A conditional expression is straightforward. We first check that the condition is a Boolean (languages vary as to whether they force the condition to be a Boolean—we shall do so, like ML or Scala does), and then check that both branches of the conditional have the same type. If so, then that common type is returned.

```
class EIf (val ec : Exp, val et : Exp, val ee : Exp) extends Exp {
  ...

  def typeOf (symt:Env[Type]) : Type = {
    if (ec.typeCheck(TBoolean,symt)) {
      val t = et.typeOf(symt)
      if (ee.typeCheck(t,symt)) {
        return t
      } else {
        terror("Branches of conditional have different types")
      }
    } else {
      terror("Condition should be Boolean")
    }
  }
}
```

Typing identifiers is where that mysterious argument to `typeOf()` comes into play. What is the type of an identifier? Well, by itself, we don't know. It depends what the identifier stands for. That's exactly the same phenomenon we encountered for evaluation. What is the value of an identifier? It depends what the identifier stands for. To solve that problem when evaluating, we use an environment that tracks the values bound to identifiers — that environment is updated when a function is applied to arguments, or during a let. To evaluate an identifier, we look up its value in the environment.

For typing, we will do something similar. When we introduce an identifier binding, we will store its type in an environment called a **symbol table**, and pass that symbol table around. When we need the type of an identifier, we look it up in the symbol table:

```
class EId (val id : String) extends Exp {
  ...

  def typeOf (symt:Env[Type]) : Type =  symt.lookup(id)
}
```

Typing functions is a multi-step process because there's a lot going on with functions. First, we need to annotate our functions with types in our surface syntax, which will propagate down to the abstract representation as an extra argument to `EFunction` holding the type of the function. Then to type check we make sure that the annotated type is a function type with the right number of parameter types, we add the parameters of the functions to the symbol table with their corresponding parameter types, and get the type of the body of the function in that new symbol table. (That will ensure that the parameters used in the body of the function will be considered at their right type.) The obtained type is compared to the annotated return type before being returned. Recursive functions are handled in exactly the same way, except that in addition the self-reference name of the function is also added to the symbol table with its annotated type, so that references to the name of the function in the body are correctly typed.

```
class EFunction (val params : List[String], val typ : Type, val body : Exp) extends Exp {
  ...

    def typeOf (symt:Env[Type]) : Type = {
      if (!typ.isFunction()) {
        terror("Function not defined with function type")
      }
      var tparams = typ.funParams()
      var tresult = typ.funResult()
      if (params.length != tparams.length) {
        terror("Wrong number of types supplied")
      }
      var new_symt = symt
      for ((p,pt) <- params.zip(tparams)) {
        new_symt = new_symt.push(p,pt)
      }
      if (body.typeCheck(tresult,new_symt)) {
        return typ
      } else {
        terror("Return type of function not same as declared")
      }
    }
}

class ERecFunction (val self: String, val params: List[String], val typ: Type, val body : Exp)
       extends Exp {
   ...

   def typeOf (symt:Env[Type]) : Type = {
      if (!typ.isFunction()) {
        terror("Function not defined with function type")
      }
      var tparams = typ.funParams()
      var tresult = typ.funResult()
      if (params.length != tparams.length) {
        terror("Wrong number of types supplied")
      }
      var new_symt = symt
      for ((p,pt) <- params.zip(tparams)) {
        new_symt = new_symt.push(p,pt)
      }
      // assume self has the declared function type
      new_symt = new_symt.push(self,typ)
      if (body.typeCheck(tresult,new_symt)) {
        return typ
      } else {
        terror("Return type of function not same as declared")
      }
    }
}
```

Function application is simple. The arguments to which the function is applied are checked to make sure that their type agrees with those expected by the function, and the result type of the function is the result type of the application.

```
class EApply (val f: Exp, val args: List[Exp]) extends Exp {
  ...

    def typeOf (symt:Env[Type]) : Type = {
```

```
    val t = f.typeOf(symt)
    if (t.isFunction()) {
      val params = t.funParams()
      if (params.length != args.length) {
        terror("Wrong number of arguments")
      } else {
        // check the argument types
        for ((pt,a) <− params.zip(args)) {
          if (!a.typeCheck(pt,symt)) {
            terror("Argument "+a+" not of expected type")
          }
        }
        return t.funResult()
      }
    } else {
      terror("Applied expression not of function type")
    }
  }
}
```

For completeness, we supply a local-bindings expression, which is type checked in what should be an obvious way once the above is internalized. Every expression to be bound is type checked and its type is associated with the name to which it is bound in the symbol table. That symbol table is used to type check the body of the `let`.

```
class ELet (val bindings : List[(String,Exp)], val ebody : Exp) extends Exp {
   ...

   def typeOf (symt:Env[Type]) : Type = {
      var new_symt = symt
      for ((n,e) <− bindings) {
        val t = e.typeOf(symt)
        new_symt = new_symt.push(n,t)
      }
      return ebody.typeOf(new_symt)
   }
}
```

And that's it! We need some support from the surface syntax: we need to annotate function with types, which means that we need a surface syntax for writing types. The sample code uses the following surface syntax for types:

```
int
```

```
  bool
  ivector
  (tfun (...) ...)
```

Where `tfun` represent a function type, and the type of a function that expects two integers and returns a boolean is `(tfun (int int) bool)`. Annotations go after the function parameters, as follows:

```
   (fun (a b) (tfun (int int) bool) (= a b))
```

Evaluating an expression (or a definition) in the shell is now a two-step process: after the surface syntax is parsed into an expression, that expression is first type checked, and if successful, evaluated to return a value. If an expression fails to type check, it is never evaluated.

Our shell has a top-level environment called `stdEnv` which holds names such as `+` bound to primitive operations. In order to be able to type check expressions involving those names, we need a top-level symbol table which associates types with those names. A definition will add a value binding to the top-level environment, and a type binding to the top-level symbol table.

The main property that our type system guarantees is the following, called a *soundness theorem*:

> For any expression e in our abstract representation, if e.`typeOf`($symt$) is T for some type T, then e.`eval`(env) $= v$ for some value v of type T.

(We need some coherence conditions about the environment *env* and type environment *symt* stating that the types in *symt* are those of the corresponding values in *env*.)

One consequence of this result is that we can in fact get rid of all the type checks that occur during evaluation! We know for a fact that as long as we have type checked our expressions, all the type checks that occur during evaluation succeed. That means we can remove them. Our implementation is not only much simpler now, but it is also potentially faster, since we don't have to repeatedly check the type of values at every step of the evaluation.