Notes on Static Types

Programming Languages

November 15, 2016

The languages we have developed until now have all been *dynamically typed*: values have types, and those types are checked during evaluation to make sure that things don't go wrong: primitive operations check the type of their arguments, `ECall` checks that the expression we supply as a function does indeed yield a function, etc.

Those checks during evaluation are sometimes considered costly. It doesn't show up so much in our interpreters or even our compilers because ultimately we're using Python to execute the code, and Python adds so much overhead that the checks are a drop in the bucket. But in more low-level execution models (say, assembly or C), those checks can make up a sizeable portion of the execution time. That's one reason why languages such as C don't perform those checks at all. Accordingly, the failure mode of C programs is to crash. In contrast, Java performs all these kind of checks, and as a consequence is safer (errors are converted to exceptions that can be recovered from) but also slower to execute. There is a tradeoff between efficiency and safety.

Can we have both, safety and efficiency? Up to a point yes, but it requires some work. The idea is to drop checks when we can convince ourselves (or more accurately, the interpreter/compiler) that those checks are unnecessary. When evaluating `(+ 1 2)`, for instance, checks for the type of arguments to the primitive operation `+` are unnecessary, since we know that the arguments are integers. In contrast, the checks in `(function (x) (+ x 1))` are necessary, because we do not know in advance what value will be passed to the function.

There are several kind of analyses we can perform on the source code to determine which checks are necessary and which are not. They differ in the precision that they allow, the scalability they offer, and the speed with which they can be performed.

A comparatively simple analysis relies on annotations from the programmer, and has the advantage of removing the need to check *any* type checking during evaluation. It takes the form of a *static type system*, and languages that implement that form of analysis are called *statically typed*.

Consider the simple S-expression surface syntax for REF, our functional language with first-class functions and reference cells. Recall that the abstract representation of this language include:

```
EValue
EId
EIf
EFunction
ECall
```

(I won't deal with `EPrimCall`, since we can hide it in functions predefined in the environment, the way we did for our compiler.) The values manipulated by our language include:

```
VInteger
VBoolean
VClosure
VRefCell
VNone
```

The idea is to associate with every expression (written in our abstract representation) in our program a *type*, which describes the class of values that the expression can evaluate to. An expression is assigned type T, intuitively, if it evaluates to values of type T.

The types we consider include:

```
TInteger
TBoolean
TNone
TFunction
TRef
TAny
```

Types `TInteger`, `TBoolean`, `TNone` are clearly the types for integers, Booleans, and the special value `VNone`. Type `TFunction(`$[T_1, \ldots, T_k]$`,`T_{body}`)` is the type of functions that take values of type $T_1, \ldots, T_k$ and return values of type T_{body}. Type `TRef(`T`)` is the type of reference cells that hold values of type T. Type `TAny` is a special type representing any value, and is used to deal with recursive anonymous functions (see below).

The implementation for types includes predicates for determining the kind of type at hand, and an `isEqual` method to check for type equality. The only subtlety with type equality is that `TAny` is considered to be equal to any other type. (That makes `TAny` usable wherever another type is expected.) Here is the implementation of types:

```
class Type (object):
    def isInteger (self):
        return False
    def isBoolean (self):
        return False
    def isFunction (self):
        return False
    def isRef (self):
        return False
    def isNone (self):
        return False
    def isAny (self):
        return False
    def isEqual (self,t):
        return False

class TInteger (Type):
    def __str__ (self):
        return "int"
    def isInteger (self):
        return True
    def isEqual (self,t):
        return (t.isInteger() or t.isAny())

class TBoolean (Type):
    def __str__ (self):
        return "bool"
    def isBoolean (self):
        return True
    def isEqual (self,t):
        return (t.isBoolean() or t.isAny())

class TNone (Type):
    def __str__ (self):
        return "none"
    def isNone (self):
        return True
    def isEqual (self,t):
        return (t.isNone() or t.isAny())

class TFunction (Type):
    def __init__ (self,params,result):
        self.params = params
        self.result = result
    def __str__ (self):
        return "(-> ({}) {})".format(" ".join([str(t) for t in self.params]),self.result)
    def isFunction (self):
        return True
    def isEqual (self,t):
        if t.isAny():
            return True
        if not t.isFunction():
            return False
        len_args = (len(self.params) == len(t.params))
        args = all([ a.isEqual(b) for (a,b) in zip(self.params,t.params)])
        return len_args and args and self.result.isEqual(t.result)
```

```
class TRef (Type):
    def __init__ (self,content):
        self.content = content
    def __str__ (self):
        return "(ref {})".format(self.content)
    def isRef (self):
        return True
    def isEqual (self,t):
        return (t.isRef() and self.content.isEqual(t.content)) or t.isAny()

class TAny (Type):
    def __str__ (self):
        return "any"
    def isAny (self):
        return True
    def isEqual (self,t):
        return True
```

It turns out we can associate such a type with every expression that can be build using our abstract representation,as long as we provide annotations on function parameters that state the types that the programmer claims the function can handle. To accomodate this, we extend the surface syntax of REF to include type annotations in functions:

$$\texttt{(function } (n_1, \ldots, n_k) \; (t_1, \ldots, t_k) \; \textit{expr}\texttt{)}$$

Here, $t_1, \ldots, t_k$ are types, written in the following abstract syntax:

```
int
bool
(-> (t1 ...) t)
(ref t)
```

For instance, `(-> (int int) int)` is the type of `+`, a function expecting two integers and yielding an integer, while `(-> ((ref int)) int)` is the type of the function that dereferences reference cells containing integers.

We introduce a *static type checking* step that occurs *before* evaluation, and that assigns a type to every expression, in such a way to guarantee that the result of evaluating said expression is always a value of the assigned type. We will flag as type errors expressions for which we cannot get that guarantee.

Type checking is implemented via a method `typecheck()` for every expression node in the abstract representation. Just like evaluation requires an environment that holds the values bound to identifiers, type checking requires a similar structure called a *type environment* recording the types associated with the various identifiers during type checking. For instance, the initial type environment holds the types associated with all the names bound in the initial environment.

Here is the implementation of type checking for the various expression nodes in our abstract representation:

```
class EValue (Exp):
    def __init__ (self,v):
        self._value = v
        self.expForm = "EValue"
    ...

    def typecheck (self,symtable):
        # type is the type of the literal in Value class
        # (using the Type classes)
        return self._value.type


class EIf (Exp):
    def __init__ (self,e1,e2,e3):
        self._cond = e1
        self._then = e2
        self._else = e3
        self.expForm = "EIf"
    ...

    def typecheck (self,symtable):
        tcond = self._cond.typecheck(symtable)
        tthen = self._then.typecheck(symtable)
        telse = self._else.typecheck(symtable)
        if not tcond.isBoolean():
            raise Exception("Type error: EIf condition should be Boolean")
        if not tthen.isEqual(telse):
            raise Exception("Type error: EIf then and else parts should be the same type
    ")
        # return the one type that is not TAny (if any)
        if tthen.isAny():
            return telse
        return tthen


class EId (Exp):
    def __init__ (self,id):
        self._id = id
        self.expForm = "EId"
    ...

    def typecheck (self,symtable):
        # type is that of the identifier in the symbol table
        for (name,typ) in reversed(symtable):
            if name == self._id:
                return typ
        raise Exception("Type error: cannot find identifier {}".format(self._id))
```

```
class ECall (Exp):
    def __init__ (self,fun,exps):
        self._fun = fun
        self._args = exps
        self.expForm = "ECall"
    ...

    def typecheck (self,symtable):
        # type is the type of the result of the function
        tfun = self._fun.typecheck(symtable)
        if not (tfun.isFunction()):
            raise Exception("Type error: non-function in ECall, got {}".format(tfun))
        # found, expected...
        if len(tfun.params) != len(self._args):
            raise Exception("Type error: wrong number of arguments in ECall, expected {}
    got {}".format(len(tfun.params),len(self._args)))
        for (t,arg) in zip(tfun.params,self._args):
            if not t.isEqual(arg.typecheck(symtable)):
                raise Exception("Type error: wrong argument in ECall, expected {} got
    {}".format(t,arg.typecheck(symtable)))
        return tfun.result


class EFunction (Exp):
    def __init__ (self,params,body,types=None,name=None):
        self._params = params
        self._body = body
        self._name = name
        self.expForm = "EFunction"
        if types and len(types) == len(params):
            self._param_types = types
        else:
            raise Exception ("Type error: wrong number of types in EFunction")
    ...

    def typecheck (self,symtable):
        if self._name:
            # recursive function, so type check under the assumption that the current
            # function returns a value of type TAny (basically, any type), and read off
            # the body type we get as the final type
            # If TAny is the final type, we've just identified an infinite loop!
            tself = [(self._name,TFunction(self._param_types,TAny()))]
            tbody = self._body.typecheck(symtable+zip(self._params,self._param_types)+
    tself)
        else:
            tbody = self._body.typecheck(zip(self._params,self._param_types) + symtable)
        return TFunction(self._param_types,tbody)
```

Everything is pretty straightforward: basic values type check and return the obvious types. Type checking an `EIf` involves making sure that the condition expression has type `TBoolean`, that the then and else parts have the same type (otherwise we cannot actually predict the type of values that `EIf` can produce since they can be produced by either branch of the `EIf`), and the result is the common type of the then and else parts. Type checking a function call involves making sure that the expression in function position has type `TFunction`, that it is given the right number of arguments, that those arguments have the expected types — the type of `ECall` is then just the result type of the function.

Type checking an `EFunction` is slightly more interesting. In the case where the function is not recursive, then we simply return a `TFunction` type where the parameter types are the ones specified when `EFunction` was created, and the result type is the type of the body under the assumption that the parameters of the function are the ones specified.

In the case of a recursive anonymous function (that is, an anonymous function that refers to itself in its own body) we need to add an entry for the current function in the symbol table when type checking the body. But what type do we give the function, since we are in the middle of trying to determine exactly what that type is? It turns out that we can give the function a type of the form `TFunction(`$[T_1, \ldots, T_k]$`, TAny())` where $T_1, \ldots, T_K$ are the types for the parameters. In other words, we assume that the current function returns a value of any type. Under that assumption, we type check the body to get type T_{body} which is the actual result type of the function, and obtain the final function type `TFunction`$[T_1, \ldots, T_k]$`,`T_{body}.

The main property of the above type system is the following *soundness theorem*:

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

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

Note what this says: if an expression type checks, then evaluating that expression will not cause an error — we get a value back. (Accessorily, that value will be of the right type.) This means that we don't actually need to do dynamic checks during execution, since we are guaranteed that evaluation will not fail.

That sounds great. And it looks like we solved the initial problem perfectly: we get efficiency (no dynamic type checks) and we get safety (we don't get error during evaluation). That's suspicious, since I suggested at the beginning that there was a trade off between efficiency and safety. What gives?

What gives is that our type system suffers from *false negatives*. It will sometimes tell you that an expression does not type check, even though evaluating that expression will not cause errors. For example:

```
((function (x y) (int bool) (+ x y)) 10 20)
```

This fails to type check (because we're passing two integers to a function that expects an integer and a Boolean, and also because we're calling `+` with an integer and a Boolean instead of two integers. At least, that's what the type checker figures. But of course, in reality, we end up calling `+` with 10 and 20, and that's perfectly fine. If we don't check types, evaluation proceeds without a hitch.

It turns out it's impossible to come up with a static type system for this language that does not suffer from false negatives. (The problem of accurately determining whether an expression evaluates without any error is an undecidable problem.) So if we go the static type system route to enforce safety and still get efficiency, we need to be okay with living in a world where the type system may reject programs that evaluate without any problems.