# Static Typing

April 2, 2020

Riccardo Pucella

# Types

## Types classify values into groups

- FUNC: types are basically subclasses of Value

## Types impact expressions and operations

- (e e1 e2 …) : e should be a function
- (+ e1 e2 …) : e1, e2, … should be numbers

## Types usually clear from context in a PL

- usually disjoint groups
- in OO, classes often create types
- object of class X considered to be of type X

# Dynamic Typing

Every value has a type

The type of a value is checked when:

- we evaluate an expression that requires a value of a specific type (EApply, EIf)
- we apply a primitive operation that requires values of specific types (+, get)

We can easily dispatch on the type of a value

- e.g., operations doing different things for values of different types

# Pros of dynamic typing

## Flexibility

- values are checked for their type right before they're being used
- when they're not used, their type doesn't really matter - so ref and vec don't care about the type of values at all

## Ease of implementation

- it comes naturally out of the implementation process

# Cons of dynamic typing

## Values are checked *all the time*

- ```
  for i in range(1,10000):
     sum =+ i
  ```
  checks that sum & i are numbers on every iteration

## Checks are done late

- when you define a function, checks within the function body are not performed until the function is called
- no guarantee that function will work
- if it does, does it work on all inputs?
- what happens if the type error happens on code delivered to the client?
  ```

# Cons of dynamic typing

```
def process (arg):
  return arg if arg != 1729 else "hello"

def test (i):
  arg = process(i)
  print(arg + 1)

>>> test(10)
11
>>> test(1729)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in test
TypeError: cannot concatenate 'str' and 'int' objects
```

# Static typing

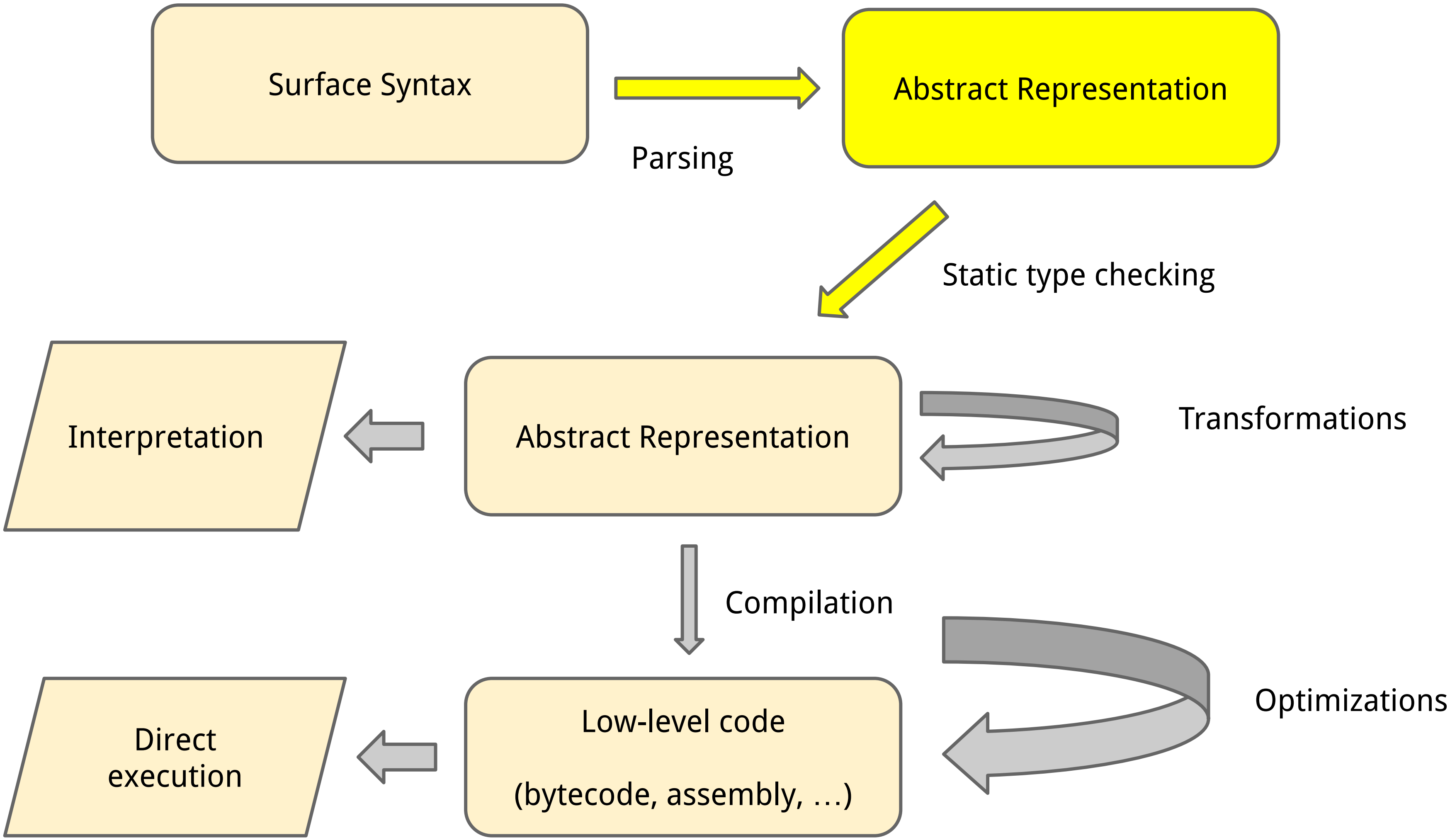
Some people are bothered by the cons above

- can we reduce the number of checks?
- can we check for all possible evaluations?
- can we somehow do the checks before running the code?

Static typing is a discipline to check types before the program is run

- *static* = without running the program
- generally checked right after parsing

# Static typing



# Approximating evaluation

Static type checking happens before the program is run

- Approximate what happens at evaluation
- Conservative approximation
  - rather reject good programs than accept bad programs
- Generally need type annotations in the code

How good an approximation you want affects:

- the type annotations required
- how difficult it is to do the actual checking
- how useful the error messages are

# Simple Types

Analyse the code, and assign a single concrete type to every expression

- intuitively, the type of values to which the expression evaluates
- all evaluations of the expression should yield a value of the given type
- if such a type does not exist, it's a type error

Example for FUNC:

- num | bool | ref(T) | vec(T) | fun(T,...)T

# Some consequences

All elements of a vector must have the same type

- otherwise expressions that pull an element out of a vector cannot be given a type, since it would depend on the index parameter

All branches of a conditional must evaluate to values of the same type

- which branch will be evaluated depends on the condition, so possibly either branch can be evaluated

…

# Typing rules

So how do we do this?

We use *typing rules* to derive the type of an expression, recursively over the structure of that expression in the abstract representation

- `EInteger(i)` has type num
- `EBoolean(i)` has type bool
- `EVector(e1, …)` has type vec(T) if `e1`, … have type T
- `ERef(e)` has type ref(T) if `e` has type T
- `EIf(c,t,e)` has type T
  if `c` has type Bool and `t,e` have type T
- `EApply(e, e1, ...)` has type T
  if `e1` has type T1, … and `e` has type fun(T1, …)T

# Typing environment

Missing: functions, and identifiers

- they both deal with names

A *type environment* assigns a type to every identifier

- similar to how environments tell you the value of every identifier)

Pass type environment **tenv** to the type checking routine

- `EId(n)` has type T if **tenv**(n) = T
- EFunction((x1 T1) ..., T, e) has type Fun(T1,...)T
  if e has type T in **tenv** extended with
  x1 having type T1, ...

# Soundness of typing rules

If expression e has type T, then when e is evaluated, e yields a value of type T

- or causes an error that has nothing to do with the type system -- e.g., /0 error

Special case: if function f has type fun(T1)T2, then applying f to a value of type T1 will yield a value of type T2

This actually can be a theorem provable of your programming language!

# Code!

```
abstract class Exp {
  ...

  def eval (env: Env[Value]) : Value
  def typeOf (tenv: Env[Type]) : Type
}


class Env[A] (val content: List[(String, A)]) {
  // parameterize Env by the type of content
  ...
}
```

# Types

```
abstract class Type {
  ...
  def compare (T: Type) : Type
}

object TNum extends Type
object TBool extends Type

class TFun (val args: List[Type], val result: Type)
                                            extends Type

class TVec (val item: Type) extends Type
class TRef (val item: Type) extends Type
```

# EInteger, EBoolean

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

  def typeOf (tenv: Env[Type]) : Type =
    TNum
}


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

  def typeOf (tenv: Env[Type]) : Type =
    TBool
}
```

# EIf

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

  def typeOf (tenv: Env[Type]) : Type = {
    val tc = ec.typeOf(tenv)
    if (!tc.isBool()) {
      typeError("Expected Boolean condition")
    }
    val tt = et.typeOf(tenv)
    val te = ee.typeOf(tenv)
    return tt.compare(te)
  }
}
```

# EId

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

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

# EApply

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

  def typeOf (tenv: Env[Type]) : Type = {
    val tfn = fn.typeOf(tenv)
    if (!tfn.isFun()) {
       typeError("Expected FUN type but received: " + tfn.toDisplay())
    }
    val targs = args.map((e:Exp) => e.typeOf(tenv))
    if (tfn.getArgs().length != targs.length) {
      typeError("Wrong number of arguments to function")
    }
    val ts = tfn.getArgs().zip(targs).map((p) => p._1.compare(p._2))
    return tfn.getResult()
  }
}
```

# EFunction

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

  def typeOf (tenv: Env[Type]) : Type = {
    var newTEnv = tenv
    for ((p, t) <- params.zip(tparams)) {
      newTEnv = newTEnv.push(p, t)
    }
    // push the specified function type as the type for identifier recName
    newTEnv = newTEnv.push(recName, new TFun(tparams, tresult))
    val tres = tresult.compare(body.typeOf(newTEnv))
    return new TFun(tparams, tres)
  }
}
```

# Surface syntax change

Functions now parse as:

```
expr ::= ...
         ( fun ( ( id typ ) ... typ ) expr )

typ ::= :num
        :bool
        ( :vec typ )
        ( :ref typ )
        ( :fun (typ ...) typ )
```

# Notes

Can do away with all the checks in expression evaluation and primitive operations!

We can add a type checking step in the shell

```
val e = parse(input)
val t = e.typeOf(tenv)
println(";; Type: " + t.toDisplay())
val v = e.eval(env)
```

We must give types to all entries in the initial environment

# Minor issue

Typing primitives in the environment

- reference primitives - need to choose a type
- vector primitives - need to choose a type

Three alternatives:

- Give different primitives for each type
  - `(ref-int 10)  (ref-bool true)  ...`
- Introduce a type *Any*
  - basically reintroduce some form of dynamic typing
- Develop a more expressive type system
  - polymorphism / generics