PURDUE UNIVERSITY®

Type Checking/Inference - Functions

Compilers: Principles And Practice

Tiark Rompf

Where Were We?

What did we learn in
the last class?

Inference Rules

```
Env |- e: T
```

Means that in the environment ‘Env’, the expression ‘e’ is of type ‘T’

This is a statement that can be **True** or **False**. This can be determined through **Inference Rules**.

Inference Rules

```
Env |- e: T
```

Means that in the environment ‘Env’, the expression ‘e’ is of type ‘T’

This is a statement that can be **True** or **False**. This can be determined through **Inference Rules**.

```
 conditions
------------ [Name of the rule]
 conclusion
```

If all conditions can be proven **True** then the conclusion is **True**.

Inference Rules

1. Lit: ‘i’ is an Int, ‘b’ is a Boolean

```
--------------------- [Int]     ------------------------- [Boolean]
 Env |- Lit(i): Int               Env |- Lit(b): Boolean

----------------------- [Unit]
 Env |- Lit(()): Unit
```

2. Unary: op is in [“+”, “-”]

```
       Env |- e: Int
--------------------------- [IntUnOp]
 Env |- Unary(op, e): Int
```

Inference Rules

3. Prim:

- op is in [“+”, “-”, “*“,”/“]
- bop is in [“==”, “!=”, “<=”, “>=”, “<”, “>”]

```
 Env |- e1: Int    Env |- e2: Int
---------------------------------- [IntOp]
   Env |- Prim(op, e1, e2): Int


 Env |- e1: Int     Env |- e2: Int
----------------------------------- [BoolOp]
 Env |- Prim(bop, e1, e2): Boolean
```

Inference Rules

4. Immutable variables

```
 Env |- e1: T1     Env,x:T1 |- e2: T2
-------------------------------------- [Let]
     Env |- Let(x, T1, e1, e2): T2
```

```
     Env(x) = T
------------------- [Ref]
 Env |- Ref(x) : T
```

Inference Rules

4. Immutable variables

```
 Env |- e1: T1     Env,x:T1 |- e2: T2
------------------------------------- [Let]
     Env |- Let(x, T1, e1, e2): T2



     Env(x) = T
------------------- [Ref]
 Env |- Ref(x) : T
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4)))
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
|- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
 ----------------------------------------------------------------[Let]
    |- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
   |- Lit(3): Int  x:Int |- Prim("==", Ref(x), Lit(4)): Boolean
-----------------------------------------------------------------[Let]
    |- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
  -----------------[Int]
    |- Lit(3): Int  x:Int |- Prim("==", Ref(x), Lit(4)): Boolean
  ------------------------------------------------------------------[Let]
     |- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
                 --------------------------------------------------[BoolOp]
    |- Lit(3): Int  x:Int |- Prim("==", Ref(x), Lit(4)): Boolean
  -----------------------------------------------------------------[Let]
     |- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
                 x:Int |- Ref(x): Int       x:Int |- Lit(4) : Int
                ------------------------------------------------[BoolOp]
     |- Lit(3): Int  x:Int |- Prim("==", Ref(x), Lit(4)): Boolean
  -----------------------------------------------------------------[Let]
      |- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
                  (x: Int)(x) = Int
               ----------------------[Ref]
                x:Int |- Ref(x): Int       x:Int |- Lit(4) : Int
               ------------------------------------------------[BoolOp]
  |- Lit(3): Int  x:Int |- Prim("==", Ref(x), Lit(4)): Boolean
----------------------------------------------------------------[Let]
   |- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Example

Prove that the following program is of type Boolean

```
val x: Int = 3; x == 4
```

```
                  (x: Int)(x) = Int
               ----------------------[Ref]-----------------------[Int]
                x:Int |- Ref(x): Int       x:Int |- Lit(4) : Int
               -------------------------------------------------------[BoolOp]
   |- Lit(3): Int  x:Int |- Prim("==", Ref(x), Lit(4)): Boolean
  ----------------------------------------------------------------[Let]
    |- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

There is no more statement to prove!! That means our initial statement was true.

Type Checking and Type Inference

Prove that the following program is of type Boolean

```
val x = 3; x == 4
```

```
Let(x, ???, Lit(3), Prim("==", Ref(x), Lit(4)))
```

Can we still do it?

Type Checking and Type Inference

Prove that the following program is of type Boolean

```
val x = 3; x == 4
```

```
   |- Lit(3): ???  x:??? |- Prim("==", Ref(x), Lit(4)): Boolean
 -----------------------------------------------------------------[Let]
    |- Let(x, ???, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Can we still do it?

Type Checking and Type Inference

Prove that the following program is of type Boolean

```
val x = 3; x == 4
```

```
 -----------------[Int]
   |- Lit(3): ???  x:??? |- Prim("==", Ref(x), Lit(4)): Boolean
 ------------------------------------------------------------------[Let]
    |- Let(x, ???, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Can we still do it? Yes, as only one rule can be applied to Lit(3).

Prove that the following program is of type Boolean

```
val x = 3; x == 4
```

```
  -----------------[Int]
    |- Lit(3): Int  x:Int |- Prim("==", Ref(x), Lit(4)): Boolean
  ------------------------------------------------------------------[Let]
     |- Let(x, Int, Lit(3), Prim("==", Ref(x), Lit(4))): Boolean
```

Can we still do it? Yes, as only one rule can be apply to Lit(3).

Type Checking and Type Inference

The type checking/inference step will be part of the semantic analyzer.

The key point to understand is that types represent an abstract value, and inference rules are the set of operations on these values.

Therefore, the implementation is going to be very similar to eval or analyze.

Type Checking and Type Inference

We add a Type field in our AST now:

```scala
abstract class Type
case class BaseType(tp: String) extends Type
val IntType = BaseType("Int")
val BoolType = BaseType("Boolean")
val UnitType = BaseType("Unit")
object UnknownType extends Type

abstract class Exp {
  // ... Position
  var tp: Type = UnknownType
  def withType(pt: Type) = { tp = pt; this }
}
```

The type checker will have to resolve the type of each node.

Type Checking and Type Inference

We are going to define two main functions:

The first is going to try to infer Type of ‘exp’ in environment ‘env’. ‘pt’ is a “suggestion” on what the type should be, but can be ignored. It returns an AST equivalent to ‘exp’ with all types resolved.

```
def typeInfer(exp: Exp, pt: Type)(env: Env): Exp
```

Inference Example

Example:

```
typeInfer(
  Let(x, UnknownType, Lit(3), Prim("==", Ref(x), Lit(4))),
  UnknownType // We don't have information at first
)(emptyEnv)
```

will return

```
Let(x, IntType,
  Lit(3),   /* tp == IntType */
  Prim("==",
    Ref(x), /* tp == IntType */
    Lit(4)  /* tp == IntType */
  )         /* tp == BoolType */
)           /* tp == BoolType */
```

Type Checking and Type Inference

The second is going to infer the Type of ‘exp’ and **verify that it comforms to type ‘pt’**. It also returns an equivalent AST with all types resolved.

```
def typeCheck(exp: Exp, pt: Type)(env: Env): Exp
```

Type Checking and Type Inference

The second is going to infer the Type of ‘exp’ and **verify that it comforms to type ‘pt’**. It also returns an equivalent AST with all types resolved.

```
def typeCheck(exp: Exp, pt: Type)(env: Env): Exp
```

We need to define what “T1 conforms to T2” means.

Type Checking and Type Inference

The second is going to infer the Type of ‘exp’ and **verify that it comforms to type ‘pt’**. It also returns an equivalent AST with all types resolved.

```
def typeCheck(exp: Exp, pt: Type)(env: Env): Exp
```

We need to define what “T1 conforms to T2” means.

T1 conforms to T2 if:

- T1 == T2, or

Type Checking and Type Inference

The second is going to infer the Type of ‘exp’ and **verify that it comforms to type ‘pt’**. It also returns an equivalent AST with all types resolved.

```
def typeCheck(exp: Exp, pt: Type)(env: Env): Exp
```

We need to define what “T1 conforms to T2” means.

T1 conforms to T2 if:

- T1 == T2, or
- T2 is UnknownType

Implementation

```
// Check if 'tp' is well-formed. For now that means that 'tp'
// is not unknown
def typeWellFormed(tp: Type)(env: Env): Type

// Check if 'tp' conforms to 'pt' and return the more precise Type
// The returned type should also be well-formed
def typeConforms(tp: Type, tp: Type)(env: Env): Type

def typeCheck(exp: Exp, pt: Type)(env: Env): Exp = {
  // First infer
  val nexp = typeInfer(exp, pt)(env)
  val rtp = typeConforms(nexp.tp, pt)(env)
  nexp.withType(rtp)
}
```

Implementation

```scala
def typeInfer(exp: Exp, pt: Type)(env: Env): Exp = exp match {
  case Lit(i: Int) => ???
  case Let(x, tp, rhs, body) => ???
  case ... => ...
}
```

Implementation

```
def typeInfer(exp: Exp, pt: Type)(env: Env): Exp = exp match {
  case Lit(i: Int) => ??? // Rule [Int]
  case Let(x, tp, rhs, body) => ??? // Rule [Let]
  case ... => ...
}
```

Implementation

```scala
def typeInfer(exp: Exp, pt: Type)(env: Env): Exp exp match {
  case Lit(i: Int) => exp.withType(IntType) // No conditions
  case Let(x, tp, rhs, body) => // Rule [Let]
    if (env.isDefined(x)) warn("reuse of variable name", exp.pos)

    // Left condition: env |- rhs: tp
    val nrhs = typeCheck(rhs, tp)(env)

    // Right condition: env, x:nrhs.tp |- body: pt (tp may be UnknownType)
    val nbody = typeCheck(body, pt)(env.withVal(x, nrhs.tp))

    // Conclusion
    Let(x, nrhs.tp, nrhs, nbody).withType(nbody.tp)
  case ... => ...
}
```

Inference Rules (cont'd)

5. if:

```
 Env |- c1: Boolean    Env |- e1: T    Env |- e2: T
---------------------------------------------------- [If]
             Env |- If(c1, e1, e2): T
```

6. Mutable variables

```
 Env |- e1: T1     Env,x:T1 |- e2: T2
-------------------------------------- [VarDec]
   Env |- VarDec(x, T1, e1, e2): T2

  Env(x) = T1     Env |- e1: T1
--------------------------------- [VarAssign]
   Env |- VarAssign(x, e1): T1
```

Inference Rules (cont'd)

7. while

```
 Env |- c1: Boolean   Env |- e1: Unit   Env |- e2: T2
------------------------------------------------------ [While]
              Env |- While(c1, e1, e2): T2
```

Interpretation With Types

```scala
abstract class Val
case class Cst(x: Any) extends Val

def eval(exp)(env: Env): Val = exp match {
  case Lit(i: Int) => Cst(i)
  case Prim(op, l, r) =>
    evalPrim(op)(eval(l)(env), eval(r)(env))
  // ...
}
def evalPrim(op: String)(l: Val, r: Val) = (op, l, r) match {
  case ("+" , Cst(x: Int), Cst(y: Int)) => Cst(x + y)
  case ("==", Cst(x: Int), Cst(y: Int)) => Cst(x == y)
  // ...
}
```

Compilation With Types

Assembly code does not have types. We need to make an "implementation" decision on how to represent the new types.

- Boolean: 0 or 1, but we still use the full register

Compilation With Types

Assembly code does not have types. We need to make an "implementation" decision on how to represent the new types.

- Boolean: 0 or 1, but we still use the full register
- Unit: There are no operations using Unit, we don't need to store it

Compilation With Types

Assembly code does not have types. We need to make an "implementation" decision on how to represent the new types.

- Boolean: 0 or 1, but we still use the full register
- Unit: There are no operations using Unit, we don't need to store it

Compilation With Types

Assembly code does not have types. We need to make an "implementation" decision on how to represent the new types.

- Boolean: 0 or 1, but we still use the full register
- Unit: There are no operations using Unit, we don't need to store it

Implementation of the operators:

```
val x = 1 == 4;
```

Compilation With Types

Assembly code does not have types. We need to make an "implementation" decision on how to represent the new types.

- Boolean: 0 or 1, but we still use the full register
- Unit: There are no operations using Unit, we don't need to store it

Implementation of the operators:

```
val x = 1 == 4;
```

We could use jumps: one branch sets 0, the other sets 1.

Compilation With Types

Assembly code does not have types. We need to make an "implementation" decision on how to represent the new types.

- Boolean: 0 or 1, but we still use the full register
- Unit: There are no operations using Unit, we don't need to store it

Implementation of the operators:

```
val x = 1 == 4;
```

We could use jumps: one branch sets 0, the other sets 1.

but X86 offers us a shortcut:

```
set<op> %al          // set %al if flags validate <op>
                     // like jump, there are: sete, setne, setl, etc.
movbq %al, %rax      // transform the byte into the full register
```

Compilation With Types

We also have to modify our compilation for the If statements.

```
def tran(exp: Exp, sp: Int)(env: Env) = exp match {
  case If(cond, tBranch, eBranch) =>
    trans(cond, sp)(env)  // now sp will contain 0 or 1
    transJumpIfTrue(sp)("if")
    // ...
```

What code would transJumpIfTrue generates? . . .

```
cmp ${regs(sp)}, $1      # INVALID syntax, only registers allowed.
je $label
```

Compilation With Types

We also have to modify our compilation for the If statements.

```
def tran(exp: Exp, sp: Int)(env: Env) = exp match {
  case If(cond, tBranch, eBranch) =>
    trans(cond, sp)(env)  // now sp will contain 0 or 1
    transJumpIfTrue(sp)("if")
    // ...
```

What code would transJumpIfTrue generates? . . .

```
cmp ${regs(sp)}, $1      # INVALID syntax, only registers allowed.
je $label
```

```
test ${regs(sp)}, ${regs(sp)}
jnz $label
```

'test S, T' sets the flags accordingly to S & T. So if 'sp' contains 1: 1 & 1 != 0 so we jump (jnz). If 'sp' contains 0: 0 & 0 == 0 so we don't jump.

Let's Add Functions

```scala
def f(x: Int) = x + 3

def g() = 2

def h(x: Int, y: Boolean): Int = {
  val z = if (y) {
    x + 1
  } else {
    x - 1
  };
  z * x
}

def k(f: Int => Int): Int = f(0)
```

Let’s Add Functions - Syntax

Let's Add Functions - Syntax

```
<type>    ::= <ident> | <type> '=>' <type>  // '=>' is right associative
            | '('[<type>[','<type>]*]')' '=>' <type>
<atom>    ::= <number> | <bool> | '()'
            | '('<simp>')'
            | <ident>
<tight>   ::= <atom>['('[<simp>[','<simp>]*]')']*
            | '{'<exp>'}'
<uatom>   ::= [<op>]<tight> // Previously atom
<simp>    ::= ... // same as before
<exp>     ::= ... // same as before
<arg>     ::= <ident>':'<type>
<prog>    ::=
   ['def'<ident>'('[<arg>[','<arg>]*]')'[':' <type>] '=' <simp>';']*
           <exp>
```

Let's Add Functions - AST

```
case class FunType(args: List[(String,Type)], rte: Type) extends Type
```

Let's Add Functions - AST

```
case class FunType(args: List[(String,Type)], rte: Type) extends Type

case class Arg(name: String, atp: Type, pos: Position)
case class FunDef(name: String, args: List[Arg], rte: Type, fbody: Exp)
        extends Exp

case class LetRec(funs: List[Exp], body: Exp) extends Exp
case class App(fun: Exp, args: List[Exp]) extends Exp
```

Let's Add Functions - Semantic

- Argument names must be distinct. Arguments behave like immutable variables.

Let's Add Functions - Semantic

- Argument names must be distinct. Arguments behave like immutable variables.
- Functions can be recursive (even mutually recursive)

Let's Add Functions - Semantic

- Argument names must be distinct. Arguments behave like immutable variables.
- Functions can be recursive (even mutually recursive)
- Function application is left associative, i.e f(1)(3) will be parsed as App(App("f", 1), 3)

Let's Add Functions - Semantic

- Argument names must be distinct. Arguments behave like immutable variables.
- Functions can be recursive (even mutually recursive)
- Function application is left associative, i.e f(1)(3) will be parsed as App(App("f", 1), 3)
- We don't allow overloading, i.e a function can not have the same name than another one even with different arguments.

Let's Add Functions - Semantic

- Argument names must be distinct. Arguments behave like immutable variables.
- Functions can be recursive (even mutually recursive)
- Function application is left associative, i.e f(1)(3) will be parsed as App(App("f", 1), 3)
- We don't allow overloading, i.e a function can not have the same name than another one even with different arguments.
- We allow functions to be stored in variables, used as parameters and returns from other functions.

Let's Add Functions - Type Checking

```
case class FunType(args: List[(String,Type)], rte: Type) extends Type
```

A function type is well-formed if all of its argument types and its return type are well-formed.

A function type 'tp' conforms to type 'pt' if all of the following hold:

1. 'pt' is a function type or UnknownType
2. 'pt' has the same number of arguments as 'tp'
3. the type of 'pt' argument #n conforms to the type of 'tp' argument #n (note the inversion)
4. the return type of 'tp' conforms to the return type of 'pt'

Let's Add Functions - Type Checking

Example:

- (Int, Boolean) => Int conforms to ??? (result: (Int, Boolean) => Int)
- Int => Int conforms to Int => Int
- Int => Int does not conform to Boolean - rule #1
- ??? => Int conforms to Int => Int (result: Int => Int)
- Int => Int does not conform to ??? => Int - rule #3
- Int => Boolean does not conform to Int => Int - rule #4
- ??? => Boolean conforms to Int => ??? (result: Int => Boolean)

Where Are We?

We looked into the formalization of type checking in our language. We discussed the implementation of the type checker.

We also started to introduce function grammar and talked about function types.

Where Are We?

We looked into the formalization of type checking in our language. We discussed the implementation of the type checker.

We also started to introduce function grammar and talked about function types.

Questions?