CS107: Type Checking/Inference and Functions

Guannan Wei
guannan.wei@tufts.edu

Jan 29, 2026

Spring 2026

Tufts University

Recap

What did we learn last time?

Recap

What did we learn last time?

- Compiling mutable variables and loops
- Syntax sugar
- Introduced type systems

Inference Rules

Typing judgments $\Gamma \vdash e : T$ assert that in the environment Γ, the expression e is of type T.

Inference Rules

Typing judgments $\Gamma \vdash e : T$ assert that in the environment Γ, the expression e is of type T.

Inference rules specify how we can form typing judgments:

$$\frac{\mathit{condition1} \qquad \mathit{condition2} \qquad \ldots}{\mathit{conclusion}}\ \textsc{name of the rule}$$

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

Inference Rules

1) Lit: i is an Int, b is a Boolean

$$\Gamma \vdash \texttt{Lit}(i) : \texttt{Int}\ \textsc{int} \qquad \Gamma \vdash \texttt{Lit}(b) : \texttt{Boolean}\ \textsc{boolean}$$

$$\Gamma \vdash \texttt{Lit}(\texttt{()}) : \texttt{Unit}\ \textsc{Unit}$$

We call inference rules without conditions **axioms**.

Inference Rules

1) Lit: i is an Int, b is a Boolean

$$\Gamma \vdash \texttt{Lit}(i) : \texttt{Int}\ \textsc{int} \qquad \Gamma \vdash \texttt{Lit}(b) : \texttt{Boolean}\ \textsc{boolean}$$

$$\Gamma \vdash \texttt{Lit}(\texttt{()}) : \texttt{Unit}\ \textsc{Unit}$$

We call inference rules without conditions **axioms**.

2) Unary: $op \in \{\texttt{"+"}, \texttt{"-"}\}$

$$\frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \texttt{Unary}(op, e) : \texttt{Int}}\ \textsc{intunop}$$

Inference Rules

3) Prim:

- $op \in \{\texttt{"+"}, \texttt{"-"}, \texttt{"*"}, \texttt{"/"}\}$
- $bop \in \{\texttt{"}=\texttt{"}, \texttt{"}\neq\texttt{"}, \texttt{"}\leqslant\texttt{"}, \texttt{"}\geqslant\texttt{"}, \texttt{"<"}, \texttt{">"}\}$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash \texttt{Prim}(op, e_1, e_2) : \texttt{Int}}\ \textsc{intop} \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash \texttt{Prim}(bop, e_1, e_2) : \texttt{Boolean}}\ \textsc{boolop}$$

Inference Rules

4) Immutable variables

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{Let}(x, T_1, e_1, e_2) : T_2}\ \text{LET} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash \texttt{Ref}(x) : T}\ \text{REF}$$

Examples

Prove that the following program is of type Boolean

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

Examples

Prove that the following program is of type Boolean

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

$$\texttt{Let}(\texttt{x},\ \texttt{Int},\ \texttt{Lit}(3),\ \texttt{Prim}(\texttt{"=="},\ \texttt{Ref}(\texttt{x}),\ \texttt{Lit}(4)))$$

Examples

Prove that the following program is of type Boolean

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

$$\emptyset \vdash \texttt{Let}(\texttt{x},\ \texttt{Int},\ \texttt{Lit}(3),\ \texttt{Prim}(\texttt{"=="},\ \texttt{Ref}(\texttt{x}),\ \texttt{Lit}(4))) : \texttt{Boolean}$$

Example

Prove that the following program is of type Boolean

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

$$\frac{}{\emptyset \vdash \texttt{Let}(\texttt{x},\ \texttt{Int},\ \texttt{Lit}(3),\ \texttt{Prim}(\texttt{"=="},\ \texttt{Ref}(\texttt{x}),\ \texttt{Lit}(4))) : \texttt{Boolean}}\ \text{LET}$$

Examples

Prove that the following program is of type Boolean

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

$$\frac{\emptyset \vdash \texttt{Lit}(3) : \texttt{Int} \qquad \texttt{x} : \texttt{Int} \vdash \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4)) : \texttt{Boolean}}{\emptyset \vdash \texttt{Let}(\texttt{x}, \texttt{Int}, \texttt{Lit}(3), \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4))) : \texttt{Boolean}}\ \text{LET}$$

Examples

Prove that the following program is of type Boolean

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

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{Lit}(3) : \texttt{Int}}\ \text{INT} \qquad \texttt{x} : \texttt{Int} \vdash \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4)) : \texttt{Boolean}}{\emptyset \vdash \texttt{Let}(\texttt{x}, \texttt{Int}, \texttt{Lit}(3), \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4))) : \texttt{Boolean}}\ \text{LET}$$

Examples

Prove that the following program is of type Boolean

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

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{Lit}(3) : \texttt{Int}}\ \text{INT} \qquad \dfrac{\texttt{x} : \texttt{Int} \vdash \texttt{Ref}(\texttt{x}) : \texttt{Int} \qquad \texttt{x} : \texttt{Int} \vdash \texttt{Lit}(4) : \texttt{Int}}{\texttt{x} : \texttt{Int} \vdash \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4)) : \texttt{Boolean}}\ \text{BOOLOP}}{\emptyset \vdash \texttt{Let}(\texttt{x}, \texttt{Int}, \texttt{Lit}(3), \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4))) : \texttt{Boolean}}\ \text{LET}$$

Examples

Prove that the following program is of type Boolean

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

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{Lit}(3) : \texttt{Int}}\,\text{INT} \qquad \dfrac{\dfrac{(\texttt{x} : \texttt{Int})(\texttt{x}) = \texttt{Int}}{\texttt{x} : \texttt{Int} \vdash \texttt{Ref}(\texttt{x}) : \texttt{Int}}\,\text{REF} \qquad \texttt{x} : \texttt{Int} \vdash \texttt{Lit}(4) : \texttt{Int}}{\texttt{x} : \texttt{Int} \vdash \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4)) : \texttt{Boolean}}\,\text{BOOLOP}}{\emptyset \vdash \texttt{Let}(\texttt{x}, \texttt{Int}, \texttt{Lit}(3), \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4))) : \texttt{Boolean}}\,\text{LET}$$

Examples

Prove that the following program is of type Boolean

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

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{Lit}(3) : \texttt{Int}}\,\text{INT} \quad \dfrac{\dfrac{(\texttt{x} : \texttt{Int})(\texttt{x}) = \texttt{Int}}{\texttt{x} : \texttt{Int} \vdash \texttt{Ref}(\texttt{x}) : \texttt{Int}}\,\text{REF} \quad \dfrac{}{\texttt{x} : \texttt{Int} \vdash \texttt{Lit}(4) : \texttt{Int}}\,\text{INT}}{\texttt{x} : \texttt{Int} \vdash \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4)) : \texttt{Boolean}}\,\text{BOOLOP}}{\emptyset \vdash \texttt{Let}(\texttt{x}, \texttt{Int}, \texttt{Lit}(3), \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4))) : \texttt{Boolean}}\,\text{LET}$$

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
```

Recall the grammar:

```
<exp>   ::= 'val' <ident> [':' <type>] '=' <simp>';'<exp>
          | ...
```

Type Checking and Type Inference

Prove that the following program is of type Boolean

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

Recall the grammar:

```
<exp>   ::= 'val' <ident> [':' <type>] '=' <simp>';'<exp>
          | ...
```

We write **???** for the placeholder/unknown type.

$$\texttt{Let}(\texttt{x},\ \texttt{???},\ \texttt{Lit}(3),\ \texttt{Prim}(\texttt{"=="},\ \texttt{Ref}(\texttt{x}),\ \texttt{Lit}(4)))$$

Can we still do it?

Type Checking and Type Inference

Prove that the following program is of type Boolean

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

We write **???** for the placeholder/unknown type.

$$\frac{\emptyset \vdash \texttt{Lit}(3) : \texttt{???} \qquad \texttt{x} : \texttt{???} \vdash \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4)) : \texttt{Boolean}}{\emptyset \vdash \texttt{Let}(\texttt{x}, \texttt{???}, \texttt{Lit}(3), \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4))) : \texttt{Boolean}}\ \text{LET}$$

Can we still do it?

Type Checking and Type Inference

Prove that the following program is of type Boolean

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

We write **???** for the placeholder/unknown type.

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{Lit}(3) : \texttt{???}}\ \text{INT} \qquad \texttt{x} : \texttt{???} \vdash \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4)) : \texttt{Boolean}}{\emptyset \vdash \texttt{Let}(\texttt{x}, \texttt{???}, \texttt{Lit}(3), \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4))) : \texttt{Boolean}}\ \text{LET}$$

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

Type Checking and Type Inference

Prove that the following program is of type Boolean

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

We write **???** for the placeholder/unknown type.

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{Lit}(3) : \texttt{Int}}\ \textsc{int} \qquad \texttt{x} : \texttt{Int} \vdash \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4)) : \texttt{Boolean}}{\emptyset \vdash \texttt{Let}(\texttt{x}, \texttt{Int}, \texttt{Lit}(3), \texttt{Prim}(\texttt{"=="}, \texttt{Ref}(\texttt{x}), \texttt{Lit}(4))) : \texttt{Boolean}}\ \textsc{let}$$

Can we still do it? Yes, as only the rule INT can be applied to Lit(3). This determines the instantiation of **???** with type `Int`.

Type Checking and Type Inference

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

Key idea: types represent abstract values, and inference rules are the set of operations on these values.

The implementation of type checking/inference will structurally similar to the interpreter `eval`.

Type Checking and Type Inference

We add a `Type` field in our AST now:

```
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 the type of `exp` in environment `env`. Type `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 conforms 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 conforms 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 conforms 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 conforms 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, pt: 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

```
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

```
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 (pt 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:

$$\frac{\Gamma \vdash c_1 : \texttt{Boolean} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{If}(c_1, e_1, e_2) : T}\ \text{IF}$$

6) Mutable variables

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{VarDec}(x, T_1, e_1, e_2) : T_2}\ \text{VARDEC}$$

$$\frac{\Gamma(x) = T_1 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \texttt{VarAssign}(x, e_1) : T_1}\ \text{VARASSIGN}$$

Inference Rules (cont'd)

7) while

$$\frac{\Gamma \vdash c_1 : \texttt{Boolean} \qquad \Gamma \vdash e_1 : \texttt{Unit} \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash \texttt{While}(c_1, e_1, e_2) : T_2}\ \textsc{while}$$

Interpretation With Types

```
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, but for the ease of implementation, we still use the full register, and just let it hold uninitialized bits.

Compilation With Types

Implementation of the operators:

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

Compilation With Types

Implementation of the operators:

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

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

Compilation With Types

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

What code would `transJumpIfTrue` generate?

Compilation With Types

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

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

What code would `transJumpIfTrue` generate?

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

What code would `transJumpIfTrue` generate?

```
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` $\neq$ `0` so we jump (jnz). If `sp` contains 0: `0 & 0` $=$ `0`, then we don't jump.

Growing the Language

What is still missing in our language?

Growing the Language

What is still missing in our language?

```
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

```
<type>   ::= <ident> | <type> '=>' <type>  // '=>' is right associative
           | '('[<type>[','<type>]*]')' '=>' <type>
<atom>   ::= <number> | <bool> | <ident> | '()'
           | '('<simp>')'
<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
```

Let's Add Functions - AST

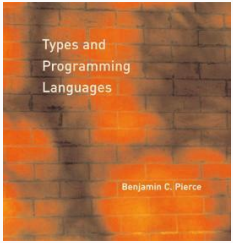
```
case class FunType(args: List[(String,Type)], rte: Type) extends Type

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
```

Example

```
                    // LetRec(List(
def f(x: Int) = { //   FunDef("f", List(Arg("x", IntType)), UnknownType,
  x + 1           //     Prim("+", Ref("x"), Lit(1))
};                //   )),
f(1)              //   App(Ref("f"), List(Lit(1)))
                  // )
```

Further Reading

- Types and Programming Languages, Benjamin C. Pierce, 2002, MIT Press
- Bidirectional Typechecking, Jana Dunfield, Neel Krishnaswami, ACM Computing Surveys, 2019
- Propositions as Types, Philip Walder, Communication of ACM, 2015. Youtube vidoe: https://www.youtube.com/watch?v=IOiZatlZtGU



Where Are We?

- We foramlized type checking/inference in our language. We discussed the implementation of the type checker.
- We started to introduce function grammar and talked about function types.

Where Are We?

- We foramlized type checking/inference in our language. We discussed the implementation of the type checker.
- We started to introduce function grammar and talked about function types.

Questions?