A Short Talk on Type Inference

— *with code!* —

preceded by a Lightning Review of the Lambda Calculus

Adrian King
30 June 2014

Part 0: The Untyped Lambda Calculus

Untyped Lambda Calculus Syntax

Only three different forms of expression:

e ::=	x	variable
	$\lambda x.\ e$	abstraction (“function”)
	$e_1\ e_2$	application (“call”)

Notation convention: in presenting syntax, a given letter (e.g., e or x) stands for any expression of the given form (in this case, e and e_i for any expression and x for any variable). e and e_i are said to *range over* expressions.

For Example

$$x\ y$$

Apply the variable *x* to the variable *y*. (Neither variable is bound in an abstraction—both are *free*.)

$$\lambda f.\ \lambda x.\ f\ (f\ x)$$

An abstraction that takes two (curried) arguments—a function *f*, and a value *x*—and applies *f* twice to *x*.

$$\lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y$$

A function of three arguments that applies its last argument to the first two.

Untyped Lambda Calculus Evaluation Rules

- A free variable evaluates to itself.

$$x \Rightarrow x$$

- If the first (function) expression in an application is an abstraction, then the application evaluates to the abstraction body with the abstraction's variable replaced by the second (argument) expression.

$$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y) \Rightarrow \lambda x.\ (\lambda y.\ y)\ ((\lambda y.\ y)\ x))$$

  You might have to rename variables to prevent conflicts.

- An abstraction evaluates to itself, except that you may (optionally) reduce an application inside the abstraction body.

$$\lambda x.\ (\lambda y.\ y)\ x \Rightarrow \lambda x.\ x$$

Substitution

- The substitution rule used in the evaluation of applications has some subtleties.
- The notation $[e_2/x]e_1$ or $[x \mapsto e_2]e_1$ means the result of substituting e_2 for x wherever x occurs in e_1, after renaming variables as necessary to avoid conflicts.
- Conflict example:

$$[z/x](\lambda z.\ x) \ \Rightarrow \ \lambda z.\ z$$

The function $\lambda z.\ x$ always returns x—substitution shouldn't turn it into the identity function! Rename it $\lambda y.\ x$.

If You're Not a Mathematician

If you're familiar with a typical programming language but not with the lambda calculus, you probably have a mental model of computation that includes things like stacks, heaps, pointers, and machine instructions.

This isn't that.

Lambda calculus evaluation is *purely symbolic*—it is defined in terms of rewriting expressions.

But in programming languages with an underlying lambda-calculus-like model*, semantics for side-effect-free expressions are often specified to be *equivalent* to expression-rewriting rules.

* aka functional languages

Amazing Fact

The untyped lambda calculus evaluation rules are sufficient to compute anything that is computable. The rules are equivalent in power to a Turing machine.

Clever Lambda Calculus Tricks

- Church numerals:

Zero	$\lambda f.\ \lambda x.\ x$
One	$\lambda f.\ \lambda x.\ f\ x$
Two	$\lambda f.\ \lambda x.\ f\ (f\ x)$

- Booleans (“if false” and “if true”):

False	$\lambda t.\ \lambda f.\ f$
True	$\lambda t.\ \lambda f.\ t$

Even More Amazing Fact

Many dialects of Lisp can be viewed as thinly disguised versions of the untyped lambda calculus, with a few bells and whistles (like built-in numbers and data structures and I/O and stuff).

I Am Lisp, and So Can You!

Almost anyone with a little coding experience can put together an untyped lambda calculus interpreter in under an hour.

Another couple of hours' work adding features is enough to develop an almost-usable (but maybe not very efficient) Lisp-like language.

So Where Did This Stuff Come From?

- Alonzo Church invented the untyped lambda calculus around 1930.
- He was *not* looking to build the first Lisp interpreter. Instead, he was trying to establish a function-based formal foundation for mathematics, as an alternative to the set-based foundations already known.
- John McCarthy invented Lisp in 1958, and Steve Russell wrote the first interpreter.

Part 1: The Simply Typed Lambda Calculus

Simply Typed Lambda Calculus Syntax

e ::=	x	variable
	$\lambda x{:}\ t.\ e$	abstraction ("function")
	$e_1\ e_2$	application ("call")
	k	constant of built-in type
t ::=	$t_1 \rightarrow t_2$	function type
	b	built-in type
b ::=	Bool | Unit | Int | *whatever*...	
k ::=	True | False | () | 0 | 1 | 2 | *etc*…	

So Where Did This Stuff Come From?

- Linguists and logicians explored the idea of *types* in languages and logic as early as the end of the 19th century.
- They were trying to eliminate nonsense utterances and logical paradoxes.
- Without types, the lambda calculus makes a lousy logic—you can use it prove anything, including obvious falsehoods.
- Church added types to the lambda calculus in the late 1930s, resulting in (more or less) the simply typed lambda calculus presented here.

Compiletime Typechecking

- Type annotations in a simply typed lambda calculus expression can be checked *before* the expression is evaluated.
- The types can then be *erased* before the program is executed, so the execution rules look just like the untyped lambda calculus.
- The type systems of most programming languages work this way.

- Lots of type theorists and language designers consider fancy type systems that mix typechecking and execution (like the dependently typed lambda calculus) to be *really scary*.

Math Geeks Used the Darnedest Symbols

- $\frac{\textbf{\textit{premise}}_1 \quad \textbf{\textit{premise}}_2}{\textbf{\textit{conclusion}}}$: A rule—you can derive *conclusion* from *premise*$_1$ and *premise*$_2$.
- **e: t**: "*e* has type *t*", or "*e* is a *t*".
- **Γ**: A context, which here means a mapping between expressions and their types, as established by the enclosing type annotations at a particular point in an expression. "*x*: *t* ∈ Γ" means Γ maps expression *x* to type *t*. "Γ, *x*: *t*" means Γ with the mapping *x*: *t* added.
- **Γ ⊢ *p***: You can read this as "In the context Γ, *p* is true" or "The context Γ implies *p*".
- Example:

$$\frac{x: t \in \Gamma}{\Gamma \vdash x: t}$$

Simply Typed Lambda Calculus Typing Rules

- Abstractions:

$$\frac{\Gamma, x: t_1 \vdash e: t_2}{\Gamma \vdash (\lambda x: t_1.\ e): t_1 \to t_2}$$

- Applications:

$$\frac{\Gamma \vdash e_1: t_1 \to t_2 \qquad \Gamma \vdash e_2: t_1}{\Gamma \vdash e_1\ e_2: t_2}$$

- Plus a whole bunch of individual rules for built-in constants and functions:

$$\frac{}{0: \text{Int}} \qquad \frac{}{\text{True: Bool}} \qquad \frac{}{+: \text{Int} \to \text{Int} \to \text{Int}}$$

yada yada yada.

The Simply Typed Lambda Calculus Is a Big Fat Pain

As a practical programming language, the simply typed lambda calculus is hopeless.

It is not sufficiently expressive to describe such commonly used constructs as sum and product types.

Its lack of genericity makes you write the same code over and over again.

It is a stop on the way to interesting type systems, but it's no place to linger.

Part 2: System F
aka
Parametric Polymorphism

Meet the Lambdas

- Vocabulary: *expression*, strictly speaking, is any form allowed in a language; *term* means an expression that can be reduced to a value. In the simply typed lambda calculus, a type is an expression but not a term.
- In the untyped lambda calculus, **terms can depend on terms**.
- In the simply typed lambda calculus, **terms are constrained by types**. This reduces the power of the calculus relative to the untyped lambda calculus—some terms cannot be typed—but makes code written in the calculus more readable to human beings.
- In System F, **terms can also depend on types**. This fixes a lot of the inexpressiveness of the simply typed lambda calculus by allowing more terms to be typed, and by making types more informative to a human reader.

System F Syntax

e ::=	x	variable
	$\lambda x{:}\ t.\ e$	abstraction (“function”)
	$e_1\ e_2$	application (“call”)
	k	constant of built-in type
	$\Lambda\alpha.\ e$	type abstraction
	$e[t]$	type application
t ::=	$t_1 \rightarrow t_2$	function type
	b	built-in type
	α	type variable
	$\forall\alpha.\ t$	type quantification

b ::= Bool | Unit | Int | *whatever*...

k ::= True | False | () | 0 | 1 | 2 | *etc*…

System F Typechecking

- As in the simply typed lambda calculus, you can erase System F types after typechecking to produce an untyped lambda calculus program.
- Two new System F typechecking rules:

$$\frac{\Gamma, \alpha \vdash e : t}{\Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\ t}$$

and:

$$\frac{\Gamma \vdash e : \forall\alpha.\ t_1}{\Gamma \vdash e[t_2] : [t_2/\alpha]t_1}$$

So Where Did This Stuff Come From?

- Jean-Yves Girard invented System F in 1972 in the context of proof theory.
- John Reynolds reinvented essentially the same system in 1974 in the context of programming language semantics.

System F Rocks!

- Polymorphic types let you declare things like sum and product types or generic functions without having to build them into your language as primitives, as you have to do with the simply typed lambda calculus.
- System F (with some restrictions and additions) is the core of the type system of languages like Haskell and Scala.

Part 3: The Dependently Typed Lambda Calculus

I want to learn the ways of the dependently typed lambda calculus and become a Jedi like my father.

In the dependently typed lambda calculus, **types can depend on terms**. This makes the dependently typed lambda calculus vastly more expressive even than System F, but the implications are too complex to explore today.

Ready, are you? What know you of ready? For eight hundred years have I trained Jedi. My own counsel will I keep on who is to be trained!

Part 1.5: Type Inference

Types That Lead to Too Much Typing

System F is pretty powerful, but programming it in the form presented here is kind of verbose. For example, the identity function:

$$\Lambda\alpha.\ \lambda(x\colon \alpha).\ x$$

is less concise than the untyped version:

$$\lambda x.\ x$$

Can we have the power of polymorphism and still write something like the untyped version?

So How Would That Work Exactly?

- First, forget about System F. We're going to explore a different species of polymorphism.
- Imagine the *simply typed lambda calculus* (with all your favorite built-in types and constants), supplemented with the ∀-quantified types from System F, but where you **never write an explicit type**. So replace the abstraction syntax:

  $$\lambda x\colon t.\ e$$

  with:

  $$\lambda x.\ e$$

  just like the untyped lambda calculus. Can we make sense of this?

What It Means to Have Types But Not Write Them

- Consider a function like:

$$\lambda n.\ n + 2$$

  Can we figure out what type this expression has? Yes, because + is defined only for type Int, and *n* is an operand of +, so *n* must be an Int. The function has type Int → Int.

- How about the identity function:

$$\lambda x.\ x$$

  This might be $\forall \alpha.\ \alpha \rightarrow \alpha$, because there's no visible restriction on the type of *x*.

But Then Again

- In this function:

$$\lambda x.\ y$$

  we don't know what *y* is, but we can see that *x* could be anything at all, because it's not mentioned in the function body. The function type must be $\forall \alpha.\ \alpha \rightarrow \beta$, where β is the type of *y*.

- Maybe *y* is unconstrained, but perhaps we could get more clues about what β is if we could step back and look at the context in which this function occurs.

A More Complex Example

- Let's give ourselves a built-in if-then-else with type

  $$\forall\alpha.\ \text{Bool} \to \alpha \to \alpha \to \alpha$$

  and look at this function:

  $$\lambda f.\ \lambda x.\ \lambda y.\ \text{if } f\ x \text{ then } x + 1 \text{ else } y$$

- $f\ x$ is a Bool, so f has type *something* → Bool.
- $x + 1$ tells us x must be an Int, so f's *something* is Int—that is, f is Int → Bool.
- y has to have the same type as $x + 1$, namely Int.
- So the overall function has type:

  $$(\text{Int} \to \text{Bool}) \to \text{Int} \to \text{Int} \to \text{Int}$$

Failure Is Always an Option

- Then there's:

$$\lambda x.\ \text{if } x \text{ then } 0 \text{ else } x + 1$$

- We see *x* is a Bool because it's the condition in the if expression, but it must also be an Int because we add one to it.
- These requirements are inconsistent, even though we could type each occurrence of *x* if it appeared in isolation.
- Sometimes our attempts to deduce a type for an expression will fail, because the expression is inherently ill-typed.

Signs Point to Yes

- By now, our intuition says there's usually a way to figure out types even if we don't state them explicitly, or at least to figure out that an expression *can't* be typed.

- But it's not always trivial to find types—sometimes we have to combine clues from different parts of an expression to narrow down the possibilities.

Systematizing the Intuition: Constraints

The key idea here is that of a *constraint*: the context in which a term appears may impose a constraint on its type, just as the context "+ 2" forces *n* to be an Int in *n* + 2.

- λ-bound variables that appear in more than one place may have multiple constraints on them, and all constraints must be satisfied for an expression to typecheck correctly.

Satisfying a System of Constraints: Unification

The Unification Algorithm

- Let's first look at how to solve a system of constraints, then we'll go back and use the resulting algorithm in the context of type inference.
- “Unification” can refer to a variety of related algorithms (depending in part on what sort of constraints can be expressed), but here we're looking at *syntactic unification of first-order terms* (which happens to the basis for the language Prolog).

First-Order Unification Terms

- A first-order unification term can be viewed as a sort of tree where the branches and some of the leaves are *constants*, but where some of the leaves may also be *variables*.
- Two constants inherently might or might not be equal to each other, but variables can be solved for to figure out what they are equal to.
- Two terms can be unified if all constants in corresponding positions in the trees are equal, and each variable can be assigned a value equal to whatever is at the corresponding position in the other tree.
- If a variable with the same name appears in more than one place, it must be assigned the same value everywhere it appears in both unification terms.

For Example

- Showing constants in black and variables in red, you can unify these two trees:

by making X equal to bar and Y equal to baz. (Imagine overlaying the trees like:

and assigning the variables to whatever winds up on top of them.)

- But you can't unify:

because gak doesn't equal foo.

For More Example

- To keep the ASCII art under control, let's write unification terms with parentheses, as in Prolog.
- So the examples from the previous slide are foo(X,baz), foo(bar,Y), and gak(bar,Y).
- Another rule: you *can* unify two variables—these terms:

  X Y

  unify with the solution X = Y. You can also unify a variable with a term that contains a variable:

  X foo(bar,Y)

  which just gives the solution X = foo(bar,Y).

Things That Can't Unify

- Two branches with different numbers of children, as in foo(X,bar) and foo(X,bar,Y).
- A variable with a term that contains the same variable, as in X and foo(X). (Such infinite terms might be useful in some places, but we won't allow them here.)

Solving a Whole System of Constraints

- The key to solving a system of constraints of the form:

  $term_1$ **must unify with** $term_2$
  $term_3$ **must unify with** $term_4$

  and so on, is that once you have an assignment for one variable, you substitute the assignment into the remaining terms, and then continue searching those terms for variable assignments.
- This might be easier to illustrate in code. Let there be code!

Data Type Declarations for the Unification Algorithm

```
module Unify (Term(..),atom,Constraint(..),unify,solve,elimVars,(??)) where

import Data.List

-- terms are parameterized by their variable representation (a):
data Term a = V a | K String [Term a]            -- V is a variable, K is a constant
atom name = K name []                            -- an atom is a constant with no children
instance (Eq a) => Eq (Term a) where
  V a1 == V a2 =                  a1 == a2
  K n1 ts1 == K n2 ts2 =          n1 == n2 && ts1 == ts2
  _ == _ =                        False
instance (Show a) => Show (Term a) where
  show (V name) =                 "?<" ++ show name ++ ">"
  show (K name []) =              name
  show (K name args) =            name ++ "(" ++ concat (intersperse "," (map show args)) ++ ")"

data Constraint a = (Term a) `MustEqual` (Term a)
instance (Show a) => Show (Constraint a) where
  show (t1 `MustEqual` t2) =     (show t1) ++ " =!= " ++ (show t2)
```

The Unification Algorithm Proper

```
unify :: (Eq a,Show a) => [Constraint a] -> Either String [Constraint a]
unify [] =
  Right []                                          -- no constraints means no constraints
unify (t1 `MustEqual` t2 : cs) | t1 == t2 =
  unify cs                                          -- t1 already equals t2 -- that's no constraint
unify (t1 @ (K n1 ts1) `MustEqual` t2 @ (K n2 ts2) : cs)
  | n1 /= n2 =
    Left ("Can't unify unequal constants: " ++ show t1 ++ " and " ++ show t2)
  | length ts1 /= length ts2 =
    Left ("Can't unify argument lists of different lengths: " ++ show t1 ++ " and " ++ show t2)
unify ((K _ ts1) `MustEqual` (K _ ts2) : cs) =  -- equal heads with args of equal length, so recurse:
  unify (correspondingArgConstraints ++ cs)
  where correspondingArgConstraints = map (uncurry MustEqual) (zip ts1 ts2)
unify (k @ (K _ _) `MustEqual` v @ (V _) : cs) =   -- swap to put V on left and handle below:
  unify (v `MustEqual` k : cs)
unify (v @ (V vName) `MustEqual` t : cs) | vName `occursIn` t =
  Left ("Infinite term would result from " ++ show v ++ " = " ++ show t)
unify (c @ ((V vName) `MustEqual` t) : cs) =
  -- we have the right definition for variable vName; substitute that definition into remaining
  -- constraints, and include it in the result:
  fmap (\newCs -> c : newCs) (unify (substitute vName t cs))
```

Helper Functions for Unification

```
occursIn :: (Eq a) => a -> (Term a) -> Bool
vName `occursIn` (V name) =     name == vName
vName `occursIn` (K _ ts) =     or (map (\t -> vName `occursIn` t) ts)

substitute :: (Eq a) => a -> (Term a) -> [Constraint a] -> [Constraint a]
substitute vName t cs =
  map substituteInConstraint cs
  where substituteInConstraint (t1 `MustEqual` t2) =
          (substituteInTerm vName t t1) `MustEqual` (substituteInTerm vName t t2)
        substituteInTerm vName t (V name) | name == vName =
          t
        substituteInTerm vName t (v @ (V _)) =
          v
        substituteInTerm vName t (K n ts) =
          K n (map (\t' -> substituteInTerm vName t t') ts)
```

Some Utility Functions for Unification

```
solve :: (Eq a,Show a) => [Constraint a] -> [(a,Term a)]
solve constraints =
  case unify constraints of
    Left errMsg ->          error (errMsg ++ "\nin constraints:\n" ++ showConstraints)
    Right cs ->             map toPair cs
  where toPair ((V v) `MustEqual` t2) =
                              (v,t2)
        toPair c =            error ("Solved constraint " ++ show c ++
                                    " should have variable on left!\nin:\n" ++ showConstraints)
        showConstraints =  concat (intersperse "\n" (map show constraints))

elimVars :: (Eq a) => [(a,Term a)] -> Term a -> Term a
elimVars solutions (thisVar @ (V v)) =
  case lookup v solutions of
    Just nextTerm -> elimVars solutions nextTerm
    Nothing -> thisVar
elimVars solutions (K name ts) =
  K name (map (elimVars solutions) ts)

(??) :: (Eq a,Show a) => [Constraint a] -> a -> Term a
constraints ?? v = elimVars (solve constraints) (V v)
```

Using Unification in Type Reconstruction

Traverse to Build Constraints, Then Solve

- The algorithm we'll use to deduce the types of expressions traverses the expression tree, potentially generating some constraints at each node, and then solves the constraint set for the whole expression.
- For example, in our example from an earlier slide:

$$\lambda f.\ \lambda x.\ \lambda y.\ \text{if } f\ x \text{ then } x + 1 \text{ else } y$$

we build constraints for visited nodes as follows:

$f\ x$	$t_f = t_x \rightarrow t_{f\ x}$
$+$	$t_+ = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
$x\ +$	$t_+ = t_x \rightarrow t_{x\ +}$
1	$t_1 = \text{Int}$
$x + 1$	$t_{x\ +} = t_1 \rightarrow t_{x\ +\ 1}$
if $f\ x$ then $x + 1$ else y	$t_{f\ x} = \text{Bool}$ $t_{x\ +\ 1} = t_y$

- Then unification gives us the same types we found earlier.

Let Polymorphism

- The language presented so far lets us omit types, but doesn't fully support polymorphism.
- In:

$$(\lambda id.\ \text{if } id \text{ True then 1 else } (id\ id)\ 2)\ (\lambda x.\ x)$$

  our algorithm tries to assign exactly one type to the variable *id*, but *id* should clearly have different concrete types at different places in this expression.

- Solution: the keyword “let” creates a sort of macro whose type is determined independently at each place the macro is used:

$$\text{let } id = \lambda x.\ x \text{ in if } id \text{ True then 1 else } (id\ id)\ 2$$

  which acts as if *id* had the type $\forall \alpha.\ \alpha \rightarrow \alpha$.

So Where Did This Stuff Come From?

- John Alan Robinson described unification in 1965. (Prolog came out in 1972.)
- The main paper on let-polymorphism is Luis Damas and Robin Milner's 1982 *Principal type-schemes for functional programs*, which describes the rules for inferring types in the programming language ML.
- The inference rules are usually known as Hindley-Milner inference. Roger Hindley, among others (including Haskell Curry), explored earlier versions of the idea.

Data Type Declarations for a Let-Polymorphic Language: Terms

```
module LetPoly (Term(..), Type(..), TypeVar(..)) where

data Term =                                          -- Const doesn't do anything but typecheck
    Var String | Lam String Term | App Term Term | Const String Type | If Term Term Term |
    Let String Term Term
  deriving (Eq)
instance Show Term where
  show (Var v) =                v
  show (Lam v body) =           "(\\" ++ v ++ ". " ++ show body ++ ")"
  show (App fun arg) =
    showMaybeParened fun ++ " " ++ showMaybeParened arg
    where showMaybeParened t @ (App _ _) =   "(" ++ show t ++ ")"
          showMaybeParened t =               show t
  show (Const name _) =       name
  show (If cond trueBranch falseBranch) =
    "(" ++ show cond ++ " ?? " ++ show trueBranch ++ " !! " ++ show falseBranch ++ ")"
  show (Let v def body) = "(let " ++ v ++ " = " ++ show def ++ " in " ++ show body ++ ")"
```

Data Type Declarations for a Let-Polymorphic Language: Types

```
data Type =                                    -- in LetType, Term is macro definition
    BoolType | IntType | FunType Type Type | LetType Term | Unknown TypeVar
  deriving (Eq)
instance Show Type where
  show BoolType =          "Bool"
  show IntType =           "Int"
  show (FunType t1 t2) =
    showMaybeParened t1 ++ " -> " ++ showMaybeParened t2
    where showMaybeParened t @ (FunType _ _) =   "(" ++ show t ++ ")"
          showMaybeParened t =                   show t
  show (LetType t) =       "[macro def: " ++ show t ++ "]"
  show (Unknown tv) =      show tv

-- Int in TypeVar is a counter to distinguish different
-- occurrences of structurally identical terms:
data TypeVar = TypeVar Int Term deriving (Eq)
instance Show TypeVar where
  show (TypeVar n term) = "<#" ++ show n ++ ": " ++ show term ++ ">"
```

Declarations for Typechecking and Type Inference

```
module Infer where

import Unify hiding (Term,Constraint)
import qualified Unify (Term,Constraint)
import LetPoly hiding (Term)
import qualified LetPoly (Term)

type Term = LetPoly.Term
type Unificand = Unify.Term TypeVar
type Constraint = Unify.Constraint TypeVar
type Env = [(String,Type)]
```

Typechecking/Type Inference: Variables, Abstractions, Applications

```
-- discover types of expressions while generating constraints on those types
typeCheck ::  Env -> Term -> Int -> (Type,[Constraint],Int)
typeCheck e (Var v) n =
  case lookup v e of
    Just (LetType def) ->        typeCheck e def n
    Just t ->                    (t,[],n)
    Nothing ->                   error ("Unknown variable " ++ v)
typeCheck e lam @ (Lam v body) n =
  let varT =                              Unknown (TypeVar n (Var v))
      (bodyT,bodyConstraints,n') =    typeCheck ((v,varT) : e) body (n + 1)
  in  (FunType varT bodyT,bodyConstraints,n')
typeCheck e app @ (App fun arg) n =
  let (funT,funConstraints,n') =       typeCheck e fun n
      (argT,argConstraints,n'') =       typeCheck e arg n'
      subconstraints =                 funConstraints ++ argConstraints
      resT =                           Unknown (TypeVar n'' app)
  in  (resT,(constrain funT (FunType argT resT)) : subconstraints,n'' + 1)
```

Typechecking/Type Inference: Constants, If, and Let

```
typeCheck e (Const _ t) n =              (t,[],n)
typeCheck e (If cond trueBranch falseBranch) n =
  let (condT,condConstraints,n') =   typeCheck e cond n
      (trueT,trueConstraints,n'') =      typeCheck e trueBranch n'
      (falseT,falseConstraints,n''') =   typeCheck e falseBranch n''
      subconstraints =                   condConstraints ++ trueConstraints ++ falseConstraints
  in  (trueT,(constrain condT BoolType) : (constrain trueT falseT) : subconstraints,n''')
typeCheck e (Let name def body) n = -- typecheck def separately: if unused, would go unchecked:
  let (_,defConstraints,n') =              typeCheck e def n
      (bodyT,bodyConstraints,n'') =   typeCheck ((name,LetType def) : e) body n'
  in  (bodyT,defConstraints ++ bodyConstraints,n'')
```

Helper Functions for Typechecking and Type Inference

```
constrain :: Type -> Type -> Constraint
constrain t1 t2 = (unificand t1) `MustEqual` (unificand t2)

unificand :: Type -> Unificand
unificand BoolType =            atom "bool"
unificand IntType =             atom "int"
unificand (FunType t1 t2) =     K "fun" [unificand t1,unificand t2]
unificand (Unknown tv) =        V tv

uType :: Unificand -> Type
uType (K "bool" []) =           BoolType
uType (K "int" []) =            IntType
uType (K "fun" [u1,u2]) =       FunType (uType u1) (uType u2)
uType (V tv) =                  Unknown tv
uType x =                       error ("Invalid unificand " ++ show x)

check :: Term -> (Type,[(TypeVar,Type,Type)])
check term =
  let (t,constraints,_) =       typeCheck [] term 1
      solutions =               solve constraints
  in  (uType (elimVars solutions (unificand t)),
         map (\(tv,u) -> (tv,uType u,uType (elimVars solutions u))) solutions)
```

Some Test Cases

```
fortyTwo =    Const "42" IntType

inc =         Const "inc" (FunType IntType IntType)

plus =        Const "+" (FunType IntType (FunType IntType IntType))

knot =        Const "not" (FunType BoolType BoolType)

eyeDee =      Lam "x" (Var "x")                          -- leaves x unconstrained

t1 =          App (App plus fortyTwo) fortyTwo           -- aka 84

t2 =          Lam "f" (
                Lam "g" (
                  Lam "x" (
                    Lam "y" (
                      If (App (App (Var "f") (Var "x")) (Var "y"))
                        (App (App plus (Var "x")) (Const "1" IntType))
                        (App (Var "g") (App (App (Var "f") (Var "x")) (Const "true" BoolType)))))))
```

Beyond Hindley-Milner

Beyond Syntactic Equality

- The sort of unification we've looked at is syntactic unification of first-order terms, where we try to find variable assignments that make pairs of terms that need to be unified *syntactically equal*.
- Languages like Haskell and Scala that let you mix explicit type declarations with inferred types also need to solve constraints that say an inferred type is *compatible with* an explicit type, where compatibility isn't just syntactic equality.

Explicit and Inferred Polymorphic Types

Let's modify an earlier example to add an explicit polymorphic type:

$$(\lambda id.\ \text{if } id \text{ True then 1 else } (id\ id)\ 2)\ ((\lambda x.\ x): (\forall \alpha.\ \alpha \rightarrow \alpha))$$

This should be allowed, because nothing in the places where *id* is used contradicts the declared type $\forall \alpha.\ \alpha \rightarrow \alpha$. But if we add the constraint that:

$$t_{id} = \forall \alpha.\ \alpha \rightarrow \alpha$$

our Hindley-Milner inference implementation will complain:

$$\begin{aligned} \text{Bool} \rightarrow \text{Bool} &\neq \forall \alpha.\ \alpha \rightarrow \alpha \\ \text{Int} \rightarrow \text{Int} &\neq \forall \alpha.\ \alpha \rightarrow \alpha \\ (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) &\neq \forall \alpha.\ \alpha \rightarrow \alpha \end{aligned}$$

Instead of syntactic type equality, we want to constrain the uses of *id* to have types that are *more specific instances* of the generic type $\forall \alpha.\ \alpha \rightarrow \alpha$.

Subtypes

- Scala (but not Haskell) has subtyping as well as parametric polymorphism.
- In addition to the sort of constraint shown on the previous slide, Scala may need to constrain a type to be a subtype or supertype of another type.

Oh, what, you expected some answers?

- I don't actually know how to solve the sorts of type constraints that appear in industrial-strength languages.
- But I'd like to learn. If you know, tell us!