Type Inference and Unification

April 18, 2011
Type Checking

Given a program where the type of every variable is known, determine whether the program is well-typed

\[ f :: \text{Bool} \rightarrow \text{Float} \]
\[ f(x) = \]
\[ \quad \text{if } x \text{ then} \]
\[ \quad \quad 3.0 \]
\[ \quad \text{else} \]
\[ \quad 2.0 \times f(\text{not } x) \]

Easy, as long as we have principal (unique, most-specific) types.
Type Checking Conditionals

class C : public A, public B { ... };
class D : public A, public B { ... };

... (x>y) ? new C() : new D() ...
Type Checking Conditionals

class C : public A, public B { ... };
class D : public A, public B { ... };

... (x>y) ? new C() : new D() ...

class C implements I1, I2 { ... };
class D implements I1, I2 { ... };

... (x>y) ? new C() : new D() ...
Type Inference (a.k.a. Type Reconstruction)

Given a program with some or all types missing, is there a way to make the program type check?

\[
f(x) = 
\begin{align*}
  \text{if } x \text{ then} \\
  &3.0 \\
  \text{else} \\
  &2.0 \ast f(\text{not } x)
\end{align*}
\]
```
int sum(const Vector<int>& v)
{
    int answer = 0;
    for (Vector<int>::const_iterator i = v.begin(); v != v.end(); ++v)
        answer += *i;
    return answer;
}
```
Inference for Local Variables

```cpp
int sum(const Vector<int>& v) {
    int answer = 0;
    for (Vector<int>::const_iterator i = v.begin(); v != v.end(); ++v)
        answer += *i;
    return answer;
}
```

```cpp
int sum(const Vector<int>& v) {
    auto answer = 0;
    for (auto i = v.begin(); v != v.end(); ++v)
        answer += *i;
    return answer;
}
```
Monomorphic Type Inference

Most abstract version:

- Insert a *type metavariable* for each variable and subexpression;
- Determine the constraints that must hold;
- Solve the constraints.
Example 1

\(((\lambda f \rightarrow f) (\lambda x \rightarrow x)) \ 3\)
Example 2

\( f \rightarrow (f \ 0) + (f \ true) \)
Example 2

\[(\lambda x \to xx) (\lambda x \to xx)\]
Hindley-Milner Polymorphism

Variables defined via let (or fun) are allowed to be polymorphic (universal quantifiers over types) if their type involves unconstrained metavariables

```latex
let id = (\x -> x)
in  (id 3, id True)
end
```
Hindley-Milner Polymorphism

Variables defined via `let` (or `fun`) are allowed to be polymorphic (universal quantifiers over types) if their type involves unconstrained metavariables.

```ml
let id = (\x -> x)

in

  (id 3, id True)

end
```

```ml
(\id -> (id 3, id True)) (\x -> x)
```
H-M Consequences

- Need to know “how polymorphic” a variable is before checking its uses.
H-M Consequences

- Need to know “how polymorphic” a variable is before checking its uses.
- Must interleave constraint generation and solving
A Practical Implementation

Metavariables as write-once variables:

```haskell
type Ty = IntTy
  | BoolTy
  | ArrowTy Ty Ty
  | PairTy Ty Ty
  | MetaTy Metavar

{-
newMetavar :: IO Metavar
metaGet :: Metavar -> IO (Maybe Ty)
metaSet :: Metavar -> Ty -> IO ()
-}
```
Expanding Definitions at Top-Level

\begin{verbatim}
expand :: Ty -> IO Ty

expand (MetaTy m) =
    do x <- metaGet m
       case x of
       Nothing -> return (MetaTy m)
       Just (MetaTy m') -> expand m'
       Just t -> return t

expand t =
    return t
\end{verbatim}

In practice, we might get $m_1 \rightarrow m_2$, $m_2 \rightarrow m_3$, \ldots, $m_n \rightarrow m_n$.

Repeatedly expanding $m_1$ is slow. How might we speed things up?
Expanding Definitions at Top-Level

```haskell
expand :: Ty -> IO Ty

expand (MetaTy m) =
    do x <- metaGet m
       case x of
          Nothing -> return (MetaTy m)
          Just (MetaTy m') -> expand m'
          Just t  -> return t

expand t =
    return t
```

In practice, we might get $m_1 \rightarrow m_2$, $m_2 \rightarrow m_3$, $\ldots$, $m_{n-1} \rightarrow m_n$. 
Expanding Definitions at Top-Level

```haskell
expand :: Ty -> IO Ty

expand (MetaTy m) =
  do x <- metaGet m
     case x of
       Nothing -> return (MetaTy m)
       Just (MetaTy m') -> expand m'
       Just t -> return t

expand t =
  return t
```

In practice, we might get $m_1 \rightarrow m_2$, $m_2 \rightarrow m_3$, \ldots, $m_{n-1} \rightarrow m_n$. Repeatedly expanding $m_1$ is slow. How might we speed things up?
Side-effecting Unification

unify :: Ty -> Ty -> IO()
unify t1 t2 =
    case (expand t1, expand t2) of
        (IntTy, IntTy) -> return ()
        (BoolTy, BoolTy) -> return ()
        (MetaTy m1, t2') ->
        (t1', MetaTy m2) ->
        (ArrowTy u1 v1, ArrowTy u2 v2) ->
        (PairTy u1 v1, PairTy u2 v2) ->
        _ -> error "Type Error"
Type Inference = Type Checking + Unification

typeOf :: Ctx -> Exp -> IO Ty

typeOf ctx (Int n) = return IntTy

typeOf ctx (Plus(e1,e2)) =
do t1 <- typeOf ctx e1
t2 <- typeOf ctx e2
unify (t1, IntTy)
unify (t2, IntTy)
return IntTy

typeOf ctx (Lambda(x,e)) =
do m <- newMetavar
  let ctx' = insert ctx x (MetaTy m)
t2 <- typeOf ctx’ e
return (ArrowTy(t1, t2)
Pitfalls: Unconstrainedness

If the type of a defined variable involves only metavariables without definitions, does it follow that the variable can be polymorphic?
Pitfalls: Unconstrainedness

If the type of a defined variable involves only metavariables without definitions, does it follow that the variable can be polymorphic?

```plaintext
foo x =
  let y = x
  in y+1
```
do r <- newIORef []

  if (x > 3) then
    writeIORef r [1,2,3]
  else
    writeIORef r ["yes", "no"]

x <- readIORef r
Complexity Results

1. Given a monomorphic expression of length $n$, determining whether the expression has a type (and if so what type) can be done in time $O(n)$. However, the type may have length $O(2^n)$.

2. Given a polymorphic expression of length $n$, determining whether the expression has a type (and if so what type) can be done in time $O(2^n)$. However, the type may have length $O(2^{2n})$.

3. In practice, type inference is $O(n)$. 
Complexity Results

1. Given a monomorphic expression of length \( n \),
   - Determining whether the expression has a type (and if so what type) can be done in time \( O(n) \).
Complexity Results

1. Given a monomorphic expression of length $n$,
   - Determining whether the expression has a type (and if so what type) can be done in time $O(n)$.
   - However, the type may have length $O(2^n)$
Complexity Results

1. Given a monomorphic expression of length $n$,
   - Determining whether the expression has a type (and if so what type) can be done in time $O(n)$.
   - However, the type may have length $O(2^n)$

2. Given a polymorphic expression of length $n$,
Complexity Results

1. Given a monomorphic expression of length $n$,
   ▶ Determining whether the expression has a type (and if so what type) can be done in time $O(n)$.
   ▶ However, the type may have length $O(2^n)$

2. Given a polymorphic expression of length $n$,
   ▶ Determining whether the expression has a type (and if so what type) can be done in time $O(2^n)$. 
Complexity Results

1. Given a monomorphic expression of length \( n \),
   - Determining whether the expression has a type (and if so what type) can be done in time \( O(n) \).
   - However, the type may have length \( O(2^n) \)

2. Given a polymorphic expression of length \( n \),
   - Determining whether the expression has a type (and if so what type) can be done in time \( O(2^n) \).
   - However, the type may have length \( O(2^{2n}) \)
Complexity Results

1. Given a monomorphic expression of length $n$,
   - Determining whether the expression has a type (and if so what type) can be done in time $O(n)$.
   - However, the type may have length $O(2^n)$

2. Given a polymorphic expression of length $n$,
   - Determining whether the expression has a type (and if so what type) can be done in time $O(2^n)$.
   - However, the type may have length $O(2^{2^n})$

3. In practice, type inference is $O(n)$.
More Polymorphism

{-# LANGUAGE RankNTypes #-}

f :: (forall a. a -> a) -> (Int, Bool)
f id = (id 3, id True)

data Tree a = Leaf a
    | Node (Tree [a])

flat :: forall a. Tree a -> [a]
flat (Leaf x) = [x]
flat (Node t) = concat (flat t)