CS653: Functional Programming
2017-18 *II*nd Semester

Type Checking and Type Inferencing

Amey Karkare
karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs653
Department of CSE, IIT Kanpur

- Project Proposals
  - Title and abstract, Due date: March 12th (EXTENDED)
  - Final Demo: During last week of classes.

Agenda

- Type Checking
- Type Inferencing
- Unification

Acknowledgements

- The slides are based on Amitabha Sanyal's notes on types.

Exercise 5

$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta)\} \vdash \lambda f\ x\ y.(f\ x, f\ y) :: \forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \alpha \to (\beta,\beta)$
$\Leftarrow$ GEN
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta)\} \vdash \lambda f\ x\ y.(f\ x, f\ y) :: (\alpha \to \beta) \to \alpha \to \alpha \to (\beta,\beta)$
$\Leftarrow$ M-ABS
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (f\ x, f\ y) :: (\beta,\beta)$
$\Leftarrow$ M-APP
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (,)(f\ x) :: \beta \to (\beta,\beta)$
and
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (f\ y) :: \beta$

Once again we shall prove the first conjunct, which is more interesting.
$\Leftarrow$ M-APP
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (,) :: \beta \to \beta \to (\beta,\beta)$
and
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (f\ x) :: \beta$

Exercise 5 (Contd)

the first conjunct is proved by INST
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (,) :: \beta \to \beta \to (\beta,\beta)$
$\Leftarrow$ INST
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash (,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta)$

and the second conjunct by M-APP
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash f :: \alpha \to \beta$
and
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha,\beta), f :: \alpha \to \beta, x :: \alpha, y :: \alpha\} \vdash x :: \alpha$

Both of which are proved by VAR.

Exercise 5 (Contd)

- Note that, given our type system,

$$\forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \alpha \to (\beta, \beta)$$

  is the *most general type* of the term $\lambda f\ x\ y.(f\ x, f\ y)$.
- In this, the types of the arguments x and y are forced to be identical.
- This makes a seemingly sensible term like

$$(\lambda f\ x\ y.(f\ x, f\ y))(\lambda x.x)\ 3\ \mathit{True}$$

  ill-typed under this type system!

The source of the problem

- This type system forces one to judge the type of a lambda body from monomorphic type assumptions regarding lambda bound variables.
- Thus all occurrence of the lambda variable in the body are forced to have the same monomorphic type. This is illustrated by the step marked in red.
- The type of $(f\ x, f\ y)$ is being judged from the assumption $f :: \alpha \to \beta$. Thus both x and y are forced to have the same type α.

Modified Type Language

- To fix this problem, we first have to change the language of types

$$\tau \rightarrow \alpha \mid \tau_1 \rightarrow \tau_2 \mid \chi\tau_1 \ldots \tau_n \mid (\tau) \quad \text{(monomorphic)} \quad (1)$$

$$\sigma \rightarrow \tau \mid \forall\alpha.\sigma \mid \sigma \rightarrow \sigma \quad \text{(polymorphic)} \quad (2)$$

- We now permit quantifiers at inner levels of a type expression.
- Thus $\forall\beta\gamma.(\forall\alpha.\alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \gamma \rightarrow (\beta, \gamma)$ is now a valid type expression.

P-APP and P-ABS

- We also introduce the following two rules instead of M-ABS and M-APP.

$$\frac{\Gamma \vdash M :: \sigma_1 \to \sigma_2 \qquad \Gamma \vdash N :: \sigma_1}{\Gamma \vdash MN :: \sigma_2} \qquad \text{(P-APP)}$$

$$\frac{\Gamma, x :: \sigma_1 \vdash M :: \sigma_2}{\Gamma \vdash \lambda x.M :: \sigma_1 \to \sigma_2} \qquad \text{(P-ABS)}$$

- $(\lambda f\ x\ y.(f\ x, f\ y))\ (\lambda x.x)\ 3\ \mathit{True}$ is now well typed.

Exercises

6. Show that:
$$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta)\}$$
$$\vdash \lambda f\ x\ y.(f\ x, f\ y) :: \forall\beta\gamma.(\forall\alpha.\alpha \to \alpha) \to \beta \to \gamma \to (\beta, \gamma)$$

Exercise 6

$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta)\}$
$\quad \vdash \lambda f\ x\ y.(f\ x, f\ y) :: \forall\beta\gamma.(\forall\alpha.\alpha \to \alpha) \to \beta \to \gamma \to (\beta, \gamma)$
$\Leftarrow$ GEN (2 times) followed by P-ABS (3 times)
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta), f :: (\forall\alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash (f\ x, f\ y) :: (\beta, \gamma)$
$\Leftarrow$ P-APP (2 times)
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta), f :: (\forall\alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash (,) :: \beta \to \gamma \to (\beta, \gamma)$
and
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta), f :: (\forall\alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash f\ x :: \beta$
and
$\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta), f :: (\forall\alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash f\ y :: \gamma$

Exercise 6

- If the judgment goes through, *f x* and *f y* would have different types.
- We just show that *f x* $:: \beta$. This follows from:
  $\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta), f :: (\forall\alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash f :: \beta \to \beta$

  and

  $\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta), f :: (\forall\alpha.\alpha \to \alpha), x :: \beta, y :: \gamma\} \vdash x :: \beta$
- The first conjunct is got by an INST followed by a VAR.
- The second conjunct directly by a VAR.

Exercises

1. Complete the above judgment and make sure that you understand it.
2. Derive the following judgment:
   $\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta), 3 :: \mathit{Int}, \mathit{True} :: \mathit{Bool}\}$
   $\vdash (\lambda f\ x\ y.(f\ x, f\ y))(\lambda x.x)\ 3\ \mathit{True} :: (\mathit{Int}, \mathit{Bool})$

Problem Solved?

- it would seem that things would be fine if we use P-ABS and P-APP instead of M-ABS and M-APP.
- However, the problem of designing an algorithm based on the above type system is open.
- Milner's language: $\lambda \rightarrow$ Curry + let

Extensions to the language: let and letrec

- The (sequential) let expression introduces a local environment.
- The expression

  `let`

  $x_1 = e_1$

  `in` e

  is evaluated as follows:
  - First the expression e_1 is evaluated in the environment surrounding the `let` expression.
  - The expression e is then evaluated in a modified environment in which the variable x_1 is bound to this value.
- If the value of x in the surrounding context is 5, the value of

```
let
        x = x+1
in 2*x
```

  is 12.

let

- The let expression can also be generalized to

```
let
        x1 = e1
        x2 = e2
        ...
        xn = en
in e
```

- This is equivalent to:

```
let
    x1 = e1
in let
            x2 = e2
                ...
                in let
                        xn = en
                    in e
```

let

We abbreviate the general let for $i = 1 \ldots n$ as:

```
let
```
$$x_i = e_i$$
```
in e
```

letrec

- `letrec` are used for recursive definitions such as

```
letrec
  fact = λn.if (n == 0) then 1 else fact(n-1)
in fact 5
```

- Note that there is no (sequential) `let` in Haskell. The Haskell `let` is actually a `letrec`.

letrec

- The `letrec` expression

```
letrec
          x = x+1
in 2*x
```

  yields an undefined value
- Because the `x` on the left and right hand side of the = represents the same `x`, and there is no (integer) value of `x` satisfying `x = x+1`.

The `let` rule

$$\frac{\Gamma_{i-1} \vdash e_i :: \sigma_i \qquad \Gamma_i = \Gamma_{i-1} \cup \{x_i :: \sigma_i\} \qquad \Gamma_n \vdash e :: \sigma}{\Gamma_0 \vdash \texttt{let}\ x_i = e_i\ \texttt{in}\ e :: \sigma}\ (\mathsf{LET})$$

Example

$$
\begin{array}{rcl}
\{3 :: \mathit{Int}, \mathit{True} :: \mathit{Bool}\} & \vdash & \mathit{let}\ \mathit{id} = \lambda x.x\ \mathit{in}\ (\mathit{id}\ 3, \mathit{id}\ \mathit{True}) :: (\mathit{Int}, \mathit{Bool}) \\
 & \Leftarrow & \text{LET} \\
\{3 :: \mathit{Int}, \mathit{True} :: \mathit{Bool}\} & \vdash & \lambda x.x :: \forall\alpha.\alpha \to \alpha \\
 & \text{and} & \\
\{3 :: \mathit{Int}, \mathit{True} :: \mathit{Bool}, \mathit{id} :: \forall\alpha.\alpha \to \alpha\} & \vdash & (\mathit{id}\ 3, \mathit{id}\ \mathit{True}) :: (\mathit{Int}, \mathit{Bool})
\end{array}
$$

Complete the rest of the proof. Also notice that the polymorphic type of id is actually being used.

The `letrec` rule

$$\frac{\Gamma \cup \{x_i :: \tau_i\} \vdash e_i :: \tau_i \qquad \sigma_i = \forall \alpha_1 \ldots \alpha_n . \tau_i, \alpha_j \notin FV(\Gamma) \qquad \Gamma \cup \{x_i :: \sigma_i\} \vdash e :: \sigma}{\Gamma \vdash \texttt{letrec}\ x_i = e_i\ \texttt{in}\ e :: \sigma}\ (\textsc{letrec})$$

- Read this rule as follows:
- Assuming a monomorphic type τ_i for x_i , suppose we can judge that e_i too has the same type τ_i.
- Further, assuming that x_i has the polymorphic type σ_i, where σ_i is an appropriate generalization of τ_i, suppose we can show that e has the type σ.
- Then the `letrec` expression also has the type σ.

Limitation

- The letrec rule too has a limitation due to which sensible programs cannot be typed.
- Consider the program:

```
let
    len = λl -> if (l == []) then 0 else (1 + (len (tail l)))
    x1 = len [1,2,3]
    in len
```

- This gives the type of `len` as [*Integer*] $\rightarrow$ *Integer*.
- This is because all occurrences of `len` in the definition part of the `letrec` have the same monomorphic type.
- The use of `len` as `len [1, 2, 3]` makes the type of `len` as [*Integer*] $\rightarrow$ *Integer*.

Limitation

- However, the following program:

```
let
    len = λl -> if (l == []) then 0 else (1 + (len (tail l)))
    x1 = len [1,2,3]
    x2 = len ['a','b','c']
    in len
```

  cannot be type-checked.

- `len` is required to be both [*Integer*] $\rightarrow$ *Integer* and [*Char*] $\rightarrow$ *Integer*!

Recap of Type rules for $\lambda \rightarrow$ Curry + let

$$\Gamma \cup \{x :: \sigma\} \vdash x :: \sigma \qquad \text{(VAR)}$$

$$\Gamma \cup \{c :: \sigma\} \vdash c :: \sigma \qquad \text{(CON)}$$

$$\frac{\Gamma \vdash M :: \sigma \qquad \sigma \leq \sigma'}{\Gamma \vdash M :: \sigma'} \qquad \text{(INST)}$$

$$\frac{\Gamma \vdash M :: \sigma \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash M :: \forall \alpha.\sigma} \qquad \text{(GEN)}$$

$$\frac{\Gamma \vdash M :: \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash N :: \tau_1}{\Gamma \vdash MN :: \tau_2} \qquad \text{(M-APP)}$$

$$\frac{\Gamma, x :: \tau_1 \vdash M :: \tau_2}{\Gamma \vdash \lambda x.M :: \tau_1 \rightarrow \tau_2} \qquad \text{(M-ABS)}$$

Recap of Type rules for $\lambda \rightarrow$ Curry + let

$$\frac{\Gamma_{i-1} \vdash e_i :: \sigma_i \qquad \Gamma_i = \Gamma_{i-1} \cup \{x_i :: \sigma_i\} \qquad \Gamma_n \vdash e :: \sigma}{\Gamma_0 \vdash \texttt{let } x_i = e_i \texttt{ in } e :: \sigma} \text{(LET)}$$

$$\frac{\Gamma \cup \{x_i :: \tau_i\} \vdash e_i :: \tau_i \qquad \sigma_i = \forall \alpha_1 \ldots \alpha_n . \tau_i, \alpha_j \notin FV(\Gamma) \qquad \Gamma \cup \{x_i :: \sigma_i\} \vdash e :: \sigma}{\Gamma \vdash \texttt{letrec } x_i = e_i \texttt{ in } e :: \sigma} \text{(LETREC)}$$

Type Inferencing

Type Inferencing – Unification

- At the heart of type inferencing is unification.
- Next: Notations and Definitions.

Term

- Each variable is a term. Variables denote by x, y, z etc.
- Each constant is a term. Constants denoted by a, b, c etc.
- If f is a n-ary function (other function symbols will be denoted by g, h etc.) and $t_1 \ldots t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is a term.

Substitution

- A substitution θ is a list of pairs $\{x_1 := t_1 \ldots x_n := t_n\}$, each pair consisting of a variable x and a term t.
- A substitution θ applied to a term t (denoted $\theta\ t$): *simultaneous substitution* of the variables $x_1, \ldots x_n$, by the corresponding type expressions $t_1, \ldots t_n$.
- Example: If $\theta = \{x := h(y), y := d\}$ and $t = f(x, g(y), c)$, then, $\theta t = f(h(y), g(d), c)$.

Unifier

- If θ is a substitution and t_1 and t_2 are terms such that $\theta t_1 = \theta t_2$, then θ is called a *unifier* of t_1 and t_2.
- Example: $\sigma_1 = \{x := a, y := c, z := b, w := c\}$ is a unifier of $f(x, b, y)$ and $f(a, z, w)$.
- σ is a *most general unifier (mgu)*, if for any other unifier σ' there exists a substitution ρ such that $\sigma' = \rho \cdot \sigma$.
  - $(\rho \cdot \sigma)t = \rho(\sigma t)$
- $\sigma'' = \{x := a, y := w, z := b\}$ is a mgu of the terms $f(x, b, y)$ and $f(a, z, w)$.
  - For $\rho = \{w := c\}$, $\sigma' = \rho \cdot \sigma''$.

The Unification Algorithm with an example

Unify $f(g(x), h(b, g(h(c, d))), y)$ and $f(g(h(w, y)), x, g(z))$.

- Have two variables mgu (the mgu so far) and ws (a working set of terms yet to be unified).

It# 1: mgu = {}
ws =
$\{< f(g(x), h(b, g(h(c, d))), y), f(g(h(w, y)), x, g(z)) >\}$

It# 2: mgu = {}
ws = $\{< g(x), g(h(w, y)) >, < h(b, g(h(c, d))), x) >, <$
$y, g(z) >\}$

It# 3: mgu = {}
ws = $\{< x, h(w, y) >, < h(b, g(h(c, d))), x) >, < y, g(z) >\}$

It# 4: mgu = <x := h(w , y)>
ws = $\{< h(b, g(h(c, d))), h(w, y)) >, < y, g(z) >\}$

The Unification Algorithm with an example

It# 5: mgu = $\{< x := h(w, y) >\}$
ws = $\{< b, w >, < g(h(c, d)), y >, < y, g(z) >\}$

It# 6: mgu = $\{< x := h(b, y) >, < w := b >\}$
ws = $\{< g(h(c, d)), y >, < y, g(z) >\}$

It# 7: mgu = $\{< x, := h(b, g(h(c, d))) >, < w := b >, < y := g(h(c, d)) >\}$
ws = $\{< g(h(c, d)), g(z) >\}$

It# 8: mgu = $\{< x := h(b, g(h(c, d))) >, < w := b >, < y := g(h(c, d)) >\}$
ws = $\{< h(c, d), z >\}$

It# 9: mgu = $\{< x := h(b, g(h(c, d))) >, < w := b >, < y := g(h(c, d)) >, < z := h(c, d) >\}$
ws = {}

The Unification Algorithm

Input: Two terms t_1 and t_2.
Output: The mgu, if it exists, else an error message

1. mgu = {}; ws = $\{< t_1, t_2 >\}$
2. While ws is not empty do:
   - 2.1 Remove a pair $\{< t, t' >\}$ from ws.
   - 2.2 If t and t' are identical variables or constants, do nothing.
   - 2.3 Else if t is a variable occurring in t' or t' is a variable occurring in t, raise an error and exit.
   - 2.4 Else if t is a variable:
     Replace all terms t'' in ws **and** in mgu by $\{t := t'\}t''$.
     mgu = mgu $\cup\{< t := t' >\}$
   - 2.5 Else if t' is a variable:
     Replace all terms t'' in ws **and** in mgu by $\{t' := t\}t''$.
     mgu = mgu $\cup\{< t' := t >\}$
   - 2.6 Else if $t = f(t_{1_1}, \ldots, t_{1_n})$ and $t' = f(t'_{1_1}, \ldots, t'_{1_n})$, then
     ws = ws $\cup\{< t_{1_1}, t'_{1_1} >, \ldots, < t_{1_n}, t'_{1_n} >\}$.
   - 2.7 Else raise an error and exit.

Exercises

Determine whether the following sets of terms are unifiable, and determine the most general unifier.

1. $\{p(a, x, f(x)), p(a, y, y)\}$
2. $\{p(x, y, z), p(u, h(v, v), u)\}$
3. $\{p(f(a), g(x)), p(y, y)\}$
4. $\{p(a, x, f(g(y))), p(z, f(z), f(u))\}$
5. $\{p(f(x, a), g(y, y), z), p(f(g(a, b), z), x, a)\}$
6. $\{p(x, x, z), p(f(a, a), y, y)\}$
7. $\{p(x, f(y, z), b), p(g(a, y), f(z, g(a, x)), b)\}$
8. $\{p(a, y, u), p(x, f(x, u), g(z, b))\}$