CS653: Functional Programming
2017-18 II^{nd} Semester

Types and Type Checking

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

Acknowledgements

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

Type systems

- A type system consist of?
  - A set of terms to be typed.
  - A set of type expressions to describe the types of the terms.
  - Type rules which enable us to make judgements such as: a term M has the type σ.

Type systems

- With a type system in place, one can define the following problems:
  - Logically prove that a term M has the type σ under the rules of the type system. This is the *type checking* problem.
  - Algorithmically show that a term M has the type σ under the type system. This is what type-checkers in a compiler do.
  - Find out a type σ for the term M under the type system. This is the *type inferencing* problem. This is what the Haskell and ML compilers do.

Type Systems

- **Terms**: Eventually we want to type a reasonable subset of Haskell. But, to start with, λ-terms augmented with constants.

  $$M \to x \mid c \mid \lambda x.M \mid M_1\ M_2 \mid (M_1)$$

  x, y and z range over variables, and c to range over constants. M and N range over terms.
- This language is called $\lambda \to$ Curry.

Type Systems

- **Type expressions:** Given by the following grammar

$$
\begin{aligned}
\tau &\rightarrow \alpha \mid \tau_1 \rightarrow \tau_2 \mid \chi\ \tau_1 \ldots \tau_n \mid (\tau) && \text{(monomorphic)} \quad (1)\\
\sigma &\rightarrow \tau \mid \forall \alpha . \sigma && \text{(polymorphic)} \quad (2)
\end{aligned}
$$

- A monomorphic type expression is either a type variable (α), a function type $\tau_1 \rightarrow \tau_2$ or a type constructor χ (like List) applied to type expressions.
- Constant types are modelled by 0-ary type constructors.
- A polymorphic type is either a monomorphic type or a type with quantified type variables.
- The quantifiers appear only at the outermost level of a type expression.

Type Systems

- Examples of monomorphic types are Int, α, $\alpha \rightarrow$Int and $\alpha \rightarrow \beta$.
- Examples of polymorphic types which are not monomorphic are $\forall\alpha.\alpha$, $\forall\alpha.\alpha \rightarrow$ *Int* and $\forall\alpha\forall\beta.\alpha \rightarrow \beta$.
- If f has the type $\forall\alpha.\alpha \rightarrow$ *Int*, it can be used in a context which requires the type:
  - Int $\rightarrow$ Int
  - Bool $\rightarrow$ Int
  - (List Int) $\rightarrow$ Int
- **NOTE:** Type variables in a Haskell type expression are implicitly quantified, i.e., the type of `foldr`:

  ```
  (a -> b -> b) -> b -> [a] -> b
  ```

  is in our notation:

  $$\forall a \forall b.(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{List } a \rightarrow b$$

Type rules

- Type rules allow us to make judgment of the form

$$\Gamma \vdash M :: \sigma$$

- Read as: from the set of assumptions Γ it can be judged that M is of the type σ.
- Assumptions have the form $x :: \sigma$ (or $c :: \sigma$).
- Read as: the variable x (or constant c) is of the type σ.
- Example:

$$\{x :: \alpha\} \vdash \lambda y.x :: \forall \beta.\beta \to \alpha$$

- From the set containing the only assumption $x :: \alpha$, one can judge that $\lambda y.x$ has the type $\forall \beta.\beta \to \alpha$.

Type Rules

- Assumptions are like symbol tables used by compilers.
- A compiler extracts information from the declarations, and inserts it in the symbol table.
- Here we use assumptions to collect information about the 'λs' and use this information to process the 'bodies'.
- Example: To show

$$\{\} \vdash \lambda x \lambda y . x :: \forall \alpha \forall \beta . \alpha \to \beta \to \alpha$$

we would show as an intermediate step that, assuming that x has the type α, $\lambda y . x$ has the type $\beta \to \alpha$. This is represented as:

$$\{x :: \alpha\} \vdash \lambda y . x :: \beta \to \alpha$$

Type Rules

- The assumption set also contains the initial assumed types of constants, built-in functions and library functions.
- Example:

$$\{+ :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}, 1 :: \mathit{Int}\} \vdash \lambda x.x + 1 :: \mathit{Int} \to \mathit{Int}$$

Type Rules

- Let J_i represent judgments. Then type rules have one of two possible forms:
  1. In the first form, J holds unconditionally.

$$J \qquad \text{(AXIOM)}$$

  2. The second form enables us to infer J from $J_1, J_2 \ldots J_n$.

$$\frac{J_1 \qquad J_2 \qquad \ldots \qquad J_n}{J} \text{ (RULE OF INFERENCE)}$$

Type rules for $\lambda \rightarrow$ Curry

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

If $x :: \sigma$ is already present in the assumption set, then we can have this fact as conclusion.

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

Similarly for constants.

Type Rules

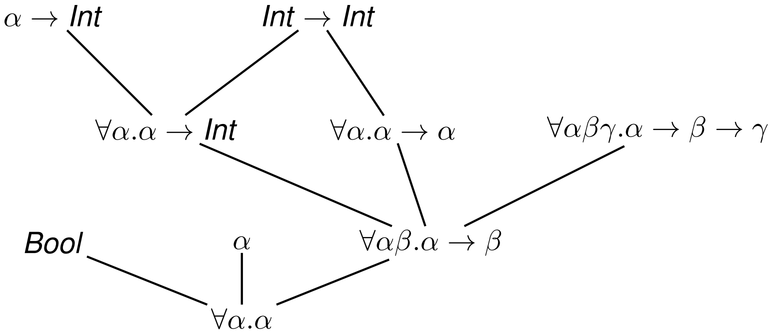
- The next rule would allow us to make inferences of the form:

$$\frac{\Gamma \vdash x :: \forall \alpha . \alpha \rightarrow \alpha}{\Gamma \vdash x :: \mathit{Int} \rightarrow \mathit{Int}}$$

- The type Int $\rightarrow$ Int is called a generic instance of $\forall \alpha . \alpha \rightarrow \alpha$.
- Intuitively, σ' is a generic instance of σ if a term having type σ can be used in any context in which a term having σ' can be used.
- Denoted $\sigma \leq \sigma'$

Type Rules

- A lattice diagram illustrating $\leq$:



Type Rules

- The type $\sigma_1 = \forall \alpha \beta . \alpha \rightarrow \beta$ has a generic instance $\sigma_2 = \forall \alpha . \alpha \rightarrow \alpha$.
- σ_2 represents a function whose argument and result have the same type.
- σ_1 represents a function, any function.
- Thus object of type σ_1 can be replaced by an object of type σ_2 without affecting the type correctness of the program.

Type Rules

- A substitution is a list of pairs denoted as $S = \{\alpha_1/\tau_1 \dots \alpha_n/\tau_n\}$.
- A substitution S applied on a type expression σ, denoted by $S(\sigma)$ involves simultaneous substitution of the variables $\alpha_1 \dots \alpha_n$, if they occur free in σ, by the corresponding type expressions $\tau_1 \dots \tau_n$.
- Definition: Let $\sigma = \forall \alpha_1 \dots \alpha_m.\tau$ and $\sigma' = \forall \beta_1 \dots \beta_n.\tau'$. Then σ' is a generic instance of σ, iff there is a substitution S acting only on $\{\alpha_1 \dots \alpha_m\}$ such that $\tau' = S(\tau)$ and no β_i is free in σ.
- Clearly, the restriction that no β_i is free in σ is needed, else we would have absurdities like

$$\alpha \to \text{Int} \leq \forall \alpha.\alpha \to \text{Int}$$

.

Rules Inst and Gen

- We are now in a position to give the next rule:

$$\frac{\Gamma \vdash M :: \sigma \qquad \sigma \leq \sigma'}{\Gamma \vdash M :: \sigma'} \qquad (\textsc{Inst})$$

- The next rule is called GEN, standing for generalization

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

- Notice the $\alpha \notin FV(\Gamma)$. Read this as “No assumption has been made about α in Γ”. The example $\lambda x \lambda y . x$ illustrates this point.

Rules M-App and M-Abs

- The final two rules are:

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

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

- The prefix **M** in **M-App** and **M-Abs** stands for monomorphic. Recall that τ ranges over monomorphic types and σ ranges over polymorphic types.

Exercises

Show the steps in type checking for each of the following terms:

1. $\{\} \vdash \lambda x.x :: \forall\alpha.\alpha \to \alpha$
2. $\{\} \vdash \lambda x\ y\ z.x\ z(y\ z) :: \forall\alpha\beta\gamma.(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$
3. $\{\} \vdash \lambda x.x :: \mathrm{Int} \to \mathrm{Int}$
4. $\lambda x.x\ x$ ::??
5. $\{(,) :: \forall\alpha\beta.\alpha \to \beta \to (\alpha, \beta)\}$
   $\vdash \quad \lambda f\ x\ y.(f\ x, f\ y) :: \forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \alpha \to (\beta, \beta)$
   Remember that `(x, y)` is actually `Pair x y`. We use the notation `( , )` both for the type constructor `Tuple` as well as the data constructor `Pair`.

Exercise 1

$$
\begin{array}{rcl}
\{\} & \vdash & \lambda x.x :: \forall\alpha.\alpha \to \alpha \\
 & \Leftarrow & \text{GEN} \\
\{\} & \vdash & \lambda x.x :: \alpha \to \alpha \\
 & \Leftarrow & \text{M-ABS} \\
\{x :: \alpha\} & \vdash & x :: \alpha \\
 & & \text{VAR}
\end{array}
$$

Exercise 2

$$
\begin{array}{rcl}
\{\} & \vdash & \lambda x\ y\ z.x\ z(y\ z) :: \\
 & & \forall\alpha\beta\gamma.(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \\
 & \Leftarrow & \text{GEN (3 times)} \\
\{\} & \vdash & \lambda x\ y\ z.x\ z(y\ z) :: \\
 & & (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \\
 & \Leftarrow & \text{M-ABS (3 times)} \\
\left\{\begin{matrix} x :: \alpha \to \beta \to \gamma, \\ y :: \alpha \to \beta, z :: \alpha \end{matrix}\right\} & \vdash & x\ z(y\ z) :: \gamma \\
 & \Leftarrow & \text{M-APP} \\
\left\{\begin{matrix} x :: \alpha \to \beta \to \gamma, \\ y :: \alpha \to \beta, z :: \alpha \end{matrix}\right\} & \vdash & x\ z :: \beta \to \gamma \\
 & \text{and} & \\
\left\{\begin{matrix} x :: \alpha \to \beta \to \gamma, \\ y :: \alpha \to \beta, z :: \alpha \end{matrix}\right\} & \vdash & (y\ z) :: \beta \\
 & & \text{consider the first conjunct only} \\
 & \Leftarrow & \text{M-APP} \\
\left\{\begin{matrix} x :: \alpha \to \beta \to \gamma, \\ y :: \alpha \to \beta, z :: \alpha \end{matrix}\right\} & \vdash & x :: \alpha \to \beta \to \gamma \\
 & \text{and} & \\
\left\{\begin{matrix} x :: \alpha \to \beta \to \gamma, \\ y :: \alpha \to \beta, z :: \alpha \end{matrix}\right\} & \vdash & z :: \alpha \\
 & & \text{VAR (once for each conjunct)}
\end{array}
$$

Exercise 3

$$\begin{array}{rcl} \{\} & \vdash & \lambda x.x :: \mathit{Int} \to \mathit{Int} \\ & \Leftarrow & \text{M-ABS} \\ \{x :: \mathit{Int}\} & \vdash & x :: \mathit{Int} \\ & & \text{VAR} \end{array}$$

The point of the above example is to show that the type system allows more than one type judgments $\forall\alpha.\alpha \to \alpha$ and Int $\to$ Int for the same term $\lambda x.x$. However, it is desirable that a type-inferencing algorithm should return a unique type (the principal type) for each term.

Exercise 4

$\lambda x.x\ x$ cannot be typed.

The reason is: assume that the type of the second occurrence of x is α. Then the type of the first occurrence of x is $\alpha \to \beta$. Since the types of the two occurrences of x must be the same, the type of the second occurrence now is $\alpha \to \beta$ and that of the first is $\alpha \to \beta \to \gamma$ and so on.

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)

- 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 $f :: \alpha \rightarrow \beta$. Thus both x and y are forced to have the same type α.