Lecture 11

Towards type inference

A powerful feature of Haskell is the automatic type inference of expressions. In the next few lectures, we will attempt to give an idea of how the type inference algorithm works. Ofcourse giving the type inference algorithm for the entire Haskell language is beyond the scope of this lecture so we take a toy example. Our aim is to give a complete type inference algorithm for an enriched version of lambda calculus, that has facilities to do integer arithmetic. Therefore, our lambda calulus expressions have, besides the other stuff we have seen, integer constants and the built in function '+'. We limit ourselves to only one operator because it is straight forward to extend our algorithm to work with other operation

11.1 Syntax of our Enriched Lambda calculus.

The syntax is given below. Here v and x stands for arbitrary variable and e_1,e_2 stands for arbitrary expressions.

$$e = ...| -1|0|1|...| + |v|e_1e_2|\lambda x.e$$

We will write the lambda calculus expression $+e1e2$ in its infix form $e1 + e2$ for ease of readability.

The haskell datatype that captures our enriched lambda calculus expression is the following

```
> module Lambda where
>
> -- | The enriched lambda calculus expression.
```

```
> data Expr = C Integer      -- ^ A constant
>           | P              -- ^ The plus operator
>           | V String       -- ^ The variable
>           | A Expr Expr    -- ^ function application
>           | L String Expr  -- ^ lambda abstraction
>       deriving (Show, Eq, Ord)
```

Clearly stuff like `A (C 2) (C 3)` are invalid expressions but we ignore this for time being. One thing that the type checker can do for us is to catch such stuff.

11.2 Types

We now want to assign types to the enriched lambda calculus that we have. As far as we are concerned the types for us are

$$t = \mathbf{Z}|\alpha|t_1 \rightarrow t_2$$

Here α is an arbitrary type variable. Again, we capture it in a Haskell datatype.

```
> data Type = INTEGER
>           | TV String
>           | TA Type Type deriving (Show, Eq, Ord)
```

11.3 Conventions

We will follow the following convention when dealing with type inference. The lambda calculus expressions will be denoted by Latin letters e, f, g etc with appropriate subscripts. We will reserve the Latin letters x, y, z and t for lambda calculus variables. Types will be represented by the Greek letter τ and σ with the letters α and β reserved for type variables.

11.4 Type specialisation

The notion of type specialisation is intuitivly clear. The type $\alpha \rightarrow \beta$ is a more general type than $\alpha \rightarrow \alpha$. We use $\sigma \leq \tau$ to denote the fact that σ is specialisation of τ. How do we formalise this notion of specialisation ? Firstly note that any constant type like for example integer cannot be specialised further. Secondly notice that a variable α can be specialised to a type τ as long as τ does not have an occurance of α in it. We will denote a variable specialisation by $\alpha \leftarrow \tau$.

When we have a set of variable specialisation we have to ensure that there is no cyclicity indirectly. We doe this as follows. We say a sequence $\Sigma = \{\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n\}$ is a consistent set of specialisation if for each i, τ_i does not contain any of the variables α_j, $1 \leq j \leq i$. Now we can define what a specialisation is. Given a consistent sequence of specialisation Σ let $\tau[\Sigma]$ denote the type obtained by substituting for variables in τ with their specialisations in Σ. Then we say that $\sigma \leq \tau$ if there is a specialisation Σ such that $\tau[\Sigma] = \sigma$. The specialisation order gives a way to compare two types. It is not a partial order but can be converted to one by appropriate quotienting. We say two types τ σ are isomorphic, denoted by $\sigma \equiv \tau$ if $\sigma \leq \tau$ and $\tau \leq \sigma$. It can be shown that $\equiv$ forms an equivalence relation on types. Let $\lceil \tau \rceil$ denote the equivalence class associated with τ then, it can be show that $\leq$ is a partial order on $\lceil \tau \rceil$.

11.5 Type environment

Recall that the value of a *closed* lambda calculus expression, i.e. a lambda calculus expression with no free variables, is completely determined. More generally, given an expression M, its value depends only on the free variables in it. Similary the type of an expression M is completely specified once all its free variables are assigned types. A *type environment* is an assignment of types to variables. So the general task is to infer the type of a lambda calculus expression M in a given type environment Γ where all the free varaibles of M have been assigned types. We will denote the type environments with with capital Greek letter Γ with appropriate subscripts if required. Some notations that we use is the following.

1. We write $x :: \tau$ to denote that the variable x has been assigned the type τ.
2. For a variable x, we use $\Gamma(x)$ to denote the type that the type environment Γ assigns to x.
3. We write $x \in \Gamma$ if Γ assigned a type for the variable x.
4. The type environment $\Gamma_1 \cup \Gamma_2$ denotes the the type environment Γ such that $\Gamma(x) = \Gamma_2(x)$ if $x \in \Gamma_2$ and $\Gamma_1(x)$ otherwise, i.e. the second type environment has a precedence.

As we described before, given a type environment Γ, the types of any lambda calculus expression whose free variables are assigned types in Γ can be infered. We use the notation $\Gamma \vdash e :: \tau$ to say that under the type environment Γ one can infer the type τ for e.

The type inference is like theorem proving: Think of infering $e :: \tau$ as proving that the expression e has type τ. Such an inference requires a set of rules which

for us will be the type inference rules. We express this inference rules in the following notation

$$\frac{\text{Premise 1}, \ldots, \text{Premise n}}{\text{conclusion}}$$

The type inference rules that we have are the following

Rule *Const* :

$$\frac{}{\Gamma \vdash n :: \mathbf{Z}}$$

where n is an arbitrary integer.

Rule *Plus* :

$$\frac{}{\Gamma \vdash + :: \mathbf{Z} \to \mathbf{Z} \to \mathbf{Z}}$$

Rule *Var* :

$$\frac{}{\Gamma \cup \{x :: \tau\} \vdash x :: \tau}$$

Rule *Apply* :

$$\frac{\Gamma \vdash f :: \sigma \to \tau, \quad \Gamma \vdash e :: \sigma}{\Gamma \vdash fe :: \tau}$$

Rule *Lambda* :

$$\frac{\Gamma \cup \{x :: \sigma\} \vdash e :: \tau}{\Gamma \vdash \lambda x.e :: \sigma \to \tau}$$

Rule *Specialise* :

$$\frac{\Gamma \vdash e :: \tau, \sigma \leq \tau}{\Gamma \vdash e :: \sigma}$$

The goal of the type inference algorithm is to infer the most general type, i.e. Given an type environment Γ and an expression e find the type τ that satisfies the following two conditions

1. $\Gamma \vdash e :: \tau$ and,
2. If $\Gamma \vdash e :: \sigma$ then $\sigma \leq \tau$.

11.6 Exercises

1. A pre-order is a relation that is both reflexive and transitive.
   - Show that the specialisation order $\leq$ defined on types is a pre-order.

- Given any pre-oder $\preceq$ define the associated relation $\simeq$ as $a \simeq b$ if $a \preceq b$ and $b \preceq a$. Prove that $\simeq$ is an equivalence class. Show that $\preceq$ can be converted into a natural partial order on the equivalence class of $\simeq$.

2. Prove that if σ and τ are two types such that $\sigma \equiv \tau$ then prove that there is a bijection between the set $Var(\sigma)$ and $Var(\tau)$ given by $\alpha_i \mapsto \beta_i$ such that $\sigma[\Sigma] = \tau$ where Σ is a specialisation $\{\alpha_i \leftarrow \beta_i | 1 \leq i \leq n\}$. In particular isomorphic types have same number of variables. (Hint: use induction on the number of variables that occur in σ and τ).