Concepts of Programming Languages

Polymorphism and Type Inference

Lecturer: Gabriele Keller
Tutor: Liam O’Connor
University of New South Wales
School of Computer Sciences & Engineering
Sydney, Australia

COMP 3161/9161 Week 8
Parametric Polymorphism

• **Example:** swap the elements of a pair (in Haskell)

\[
\text{swap} \ (x, \ y) = (y, \ x)
\]

• **What is swap’s type?**

★ In Haskell:

\[
\text{swap} :: (a, b) \to (b, a)
\]

★ in MinHs:

\[
\begin{align*}
\text{letfun swapIntBool} & :: (\text{Int}, \text{Bool}) \to (\text{Bool}, \text{Int}) \ \text{pair} = \\
& \quad (\text{snd \ pair}, \ \text{fst \ pair}) \\
\text{letfun swapBoolInt} & :: (\text{Bool}, \text{Int}) \to (\text{Int}, \text{Bool}) \ \text{pair} = \\
& \quad (\text{snd \ pair}, \ \text{fst \ pair}) \\
\end{align*}
\]

obviously, not the way to go!
Parametric Polymorphism in MinHs

- **Parametric polymorphism:**

  ```haskell
  letfun swap :: (a, b) -> (b, a) pair =
  (snd pair, fst pair)
  ```

  ★ **a and b are type variables**

- **Using a polymorphic function:**

  ★ when a polymorphic function is applied to a concrete value, the type variables are **instantiated**:

  ```haskell
  swap (1, True)
  ```

  ★ instantiates type variable

  ‣ a to Int
  ‣ b to Bool

Tuesday, 30 April 13
Parametric Polymorphism

- Assuming explicit typing
  - Introduction of type variables needs to be explicit
  - Instantiation of type variables needs to be explicit
- Type abstraction

```plaintext
Type a in
  Type b in
  letfun swap :: ((a, b) -> (b, a)) pair =
    (snd pair, fst pair)
```

- Type instantiation

```plaintext
inst (Type a in
  inst (Type b in
    letfun swap :: ((a, b) -> (b, a)) pair =
      (snd pair, fst pair)
    Bool)
  Int)
```

evaluates to a monomorphic function:

```plaintext
letfun swap :: (Int, Bool) -> (Bool, Int) pair =
  (snd pair, fst pair)
```
Parametric Polymorphism

• What is the type of this function?

```haskell
letfun swap :: ((a, b) -> (b, a)) pair = (snd pair, fst pair)
```

• Universal quantification:

★ it is $\forall a. \forall b. (a, b) \rightarrow (b, a)$

★ written in Haskell (leading `forall` optional)

```haskell
forall a b. (a, b) -> (b, a)
```
Polymorphic MinHS - Concrete Syntax

**Polytypes**
\[ \sigma ::= \tau \mid \forall t. \sigma \]

**Monotypes**
\[ \tau ::= \text{Bool} \mid \text{Int} \mid \tau_1 \to \tau_2 \]

**Expressions**
\[ e ::= v \]
\[ \mid \text{inst} \ (e, \tau) \]
\[ \mid \text{letfun} \ id_1 :: (\tau_1 \to \tau_2) \ id_2 = e \]
\[ \mid \text{Type} \ t \ \text{in} \ e \mid \ldots \]

*only monotypes here!*
Polymorphic MinHS

- Valid Types:
  - make sure we have no free type variables

\[ \begin{align*}
& t \in \Delta \\
\implies & \Delta \vdash t \text{ ok} \\
\implies & \Delta \vdash \text{Bool ok} \\
\implies & \Delta \vdash \text{Int ok} \\
\implies & \Delta \vdash \tau_1 \text{ ok} \\
\implies & \Delta \vdash \tau_1 \rightarrow \tau_2 \text{ ok} \\
\implies & \Delta \vdash \sigma \text{ ok} \\
\implies & \Delta \vdash \forall t . \sigma \text{ okP} \\
\implies & \Delta \vdash \sigma \text{ okP}
\end{align*} \]
Polymorphic MinHS

- Typing rules

\[
\begin{align*}
\Delta \cup \{t\}, \Gamma \vdash e : \sigma \quad & t \notin \Delta \\
\Delta, \Gamma \vdash \text{Type } (t.e) : \forall t.\sigma \\
\Delta, \Gamma \vdash e : \forall t.\sigma \quad & \Delta \vdash \tau \text{ ok} \\
\Delta, \Gamma \vdash \text{Inst } e \ \tau : \sigma[t:=\tau]
\end{align*}
\]
Polymorphic MinHS

- Dynamic Semantics

\[
\begin{align*}
    e & \xrightarrow{M} e' \\
    \text{Inst } e \tau & \xrightarrow{M} \text{Inst } e' \tau \\
    \text{Inst}(\text{Type}(t.e)) \tau & \xrightarrow{M} e[t:= \tau]
\end{align*}
\]
Polymorphic MinHs

• Polymorphic MinHs with
  ★ explicit introduction of type variables:
    ‣ Type a in letfun id :: (a -> a) x = x :: ∀ a.a → a

★ explicit instantiation of type variables:
  ‣ inst(Type a in letfun id : (a->a) x = x, Bool) :: Bool → Bool
Polymorphic MinHs

• Polymorphic functions are **not first class citizens** in MinHs, because we only allow polytypes at the top level
  ★ we can’t have a polymorphic function as return value
  ★ we can’t have a function that demands a polymorphic function as argument value

• This restriction is not actually necessary for **explicitly typed** MinHs, since type checking would still work without (but it would be a problem for type inference)
Polymorphic MinHs

• For which of the following types can we write total, terminating MinHs functions? (excluding error)

  ★ ∀ a. ∀ b. (a * b) → (b * a)  ✔
  ★ ∀ a. ∀ b. (a + b) → (b + a)  ✔
  ★ ∀ a. ∀ b. (a * b) → a  ✔
  ★ ∀ a. ∀ b. (a + b) → a  ✗
  ★ ∀ a. ∀ b. (a → b) → (b → a)  ✗
  ★ ∀ a. ∀ b. ∀ c. ((a → c) * (b → c)) → (a+b → c)  ✔

• The type constructors +, *, and → correspond to the logical operators ∨, ∧ and ⇒. Types to theorems, and terminating programs to (constructive) proofs, and the type checker to a proof checker.
Principal Type

• Explicitly typed polymorphic languages are awkward to use
• We want the compiler to infer the type of an expression for us.
• What is the type of this function?

\[
\text{letfun } f \ x = (\text{fst } x) + 1
\]

• Possible types

(1) \( \text{Int}^* \text{Int} \to \text{Int} \)
(2) \( \text{Int}^* \text{Bool} \to \text{Int} \)
(3) \( \text{Int}^* (\text{Int} \to (\text{Int} + \text{Bool})) \to \text{Int} \)
(4) \( \forall a. \text{Int}^* a \to \text{Int} \)

• Types (1) - (3) are instances of type (4)
Principal Type

• We write $\tau' \leq \tau$ if $\tau'$ is less general than $\tau$, that is $\tau'$ is an instance of $\tau$

$\star \text{Int} \times \text{Int} \to \text{Int} \leq \forall a. \text{Int} \times a \to \text{Int}$

$\star \forall a. \text{Int} \times a \to \text{Int} \leq \forall a. \forall b. b \times a \to b$

$\star \forall a. \text{Int} \times a \to \text{Int} \leq \forall a. \forall a. a \times a \to a$

$\star \forall a. a \times a \to a \leq \forall a. \forall b. b \times a \to b$

• We are interested in the most general type $\tau$ of the expression $e$ such that $e: \tau'$ implies $\tau' \leq \tau$

• This is called the principal type of the expression
Implicitly Typed MinHs

• MinHs with the following changes:

★ no type annotations for functions and type constructors (sum & product type)
★ roll, unroll, rec not part of the language
  ‣ not possible with implicit typing
★ no explicit type abstraction and instantiation
  ‣ Type and inst not part of the language
★ Types of the build-in functions are part of the environment:
  ‣ \( \Gamma = \{\oplus: \text{Int} \Rightarrow \text{Int} \Rightarrow \text{Int}, \text{fst}: \forall a. \forall b.(a * b) \Rightarrow a, \ldots\}\)
★ no overloading yet, e.g., \(==\) still only compares integers,
Typing Rules

• Application, if-expression, variable and product rules stay the same:

\[
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}
\]

\[\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \text{Apply } e_1 \ e_2 : \tau_2}\]

\[\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{Pair } e_1 \ e_2 : \tau_1 \times \tau_2}\]

\[\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \text{If } t_1 \ t_2 \ t_3 : \tau}\]
Typing Rules

• Functions:

\[
\Gamma \cup \{f : \tau_1 \rightarrow \tau_2, x : \tau_1 \} \vdash e : \tau_2
\]

\[
\Gamma \vdash \text{Letfun}(f.x.e) : \tau_1 \rightarrow \tau_2
\]

• Inr and Inl introduce free type variables:

\[
\Gamma \vdash e_1 : \tau_1
\]

\[
\Gamma \vdash \text{Inl} \ e_1 : \tau_1 + \tau_2
\]

\[
\Gamma \vdash e_2 : \tau_2
\]

\[
\Gamma \vdash \text{Inr} \ e_2 : \tau_1 + \tau_2
\]

could be any type
Typing Rules

- $\forall$ - introduction and elimination

\[
\frac{
\Gamma \vdash e : \forall t. \tau
}{
\Gamma \vdash e : \tau [t:=\tau']
}\]

\[
\frac{
\Gamma \vdash e : \tau \quad t \notin \text{TypeVars}(\Gamma)
}{
\Gamma \vdash e : \forall t. \tau
}\]
Typing Rules

• Do the typing rules describe a type inference algorithm?
  ★ are they syntax directed?
    ‣ no - ∀-introduction can always be applied
  ★ can we view Γ and the expression as input, the type as output?
    ‣ no - ∀-elimination rule may instantiate an expression to the wrong type!
    ‣ no - function rule may introduce wrong type for x and f in environment

Γ ⊢ Fst: ∀a.∀b.(a * b) → a

Γ ⊢ Fst: (Bool*Bool) → Bool

Γ ⊢ Pair 1 True: (Int*Bool)

Γ ⊢ Apply Fst (Pair 1 True):

Γ ⊢ Letfun(f.x. (Plus x 1)):
Type Inference Algorithm

• Idea

★ delay the instantiation of type variables until necessary

★ “merge” required and computed argument type

★ replace ∀-quantified variables by free, fresh variables

\[
\Gamma \vdash \text{Fst}: \forall a. \forall b.(a \times b) \rightarrow a \\
\Gamma \vdash \text{Fst}: (x \times y) \rightarrow x \\
\Gamma \vdash \text{Pair} \ 1 \ \text{True}: \ (\text{Bool} \times \text{Int}) \\
\Gamma \vdash \text{Apply} \ \text{Fst} \ (\text{Pair} \ 1 \ \text{True}):
\]

? 

\[(x \times y) \rightarrow x \quad [y := \text{Int}, x := \text{Bool}] = (\text{Bool} \times \text{Int}) \rightarrow z[z := \text{Int}]\]

type of function we have  type of function we need
Type Inference Algorithm

- In some cases, it is necessary to substitute variables on both sides:

\[(\text{Bool} \times x)[y := \text{Int}, x := \text{Bool}] \overset{?}{=} (y \times \text{Int})[y := \text{Int}, x := \text{Bool}]\]

or to replace variables with other variables

\[(x \times x)[y := x] \overset{?}{=} (x \times y)[y := x]\]
Unification

• A substitution $S$, with $S \tau = S \tau'$ is called a unifier of $\tau$ and $\tau'$

• For the algorithm, we need the most general unifier (mgu)

• We write $\tau_1 \sim_S \tau_2$ if $S$ is an mgu of $\tau_1$ and $\tau_2$

• Examples:

★ are there mgu’s for the following pairs of types?

$(a * (a * a)) = (b * c)$

$\text{Int} = \text{Bool}$

$(a * (a * a)) = ((a * a) * a)$
Type Inference Algorithm

• Now back to our type inference algorithm:

\[
\frac{x : \forall a_1. \ldots \forall a_n. \tau \in \Gamma}{\Gamma \vdash x : \tau[a_1 := \beta_1, \ldots, a_n := \beta_n]} \quad \beta_i \text{ fresh}
\]

\[
\frac{T \Gamma \vdash e_1 : \tau_1 \quad T_1 T \Gamma \vdash e_2 : \tau_2 \quad T_1 \tau_1 \sim U \tau_2 \rightarrow \alpha}{UT_1 T \Gamma \vdash \text{Apply } e_1 e_2 : U \alpha} \quad \alpha \text{ fresh}
\]

\[
\frac{T(\Gamma \cup \{x : \alpha_1\} \cup \{f : \alpha_2\}) \vdash e : \tau \quad T \alpha_2 \sim U T \alpha_1 \rightarrow \tau}{UT \Gamma \vdash \text{Letfun } (f.x.e) : U(T \alpha_1 \rightarrow \tau)} \quad \alpha \text{ fresh}
\]

• Note
  • the rules are syntax directed
  • the environment and expression are input & unifier and type are output
Type Inference

- Application example:

\[
T \Gamma \vdash e_1 : \tau_1 \\
T_1 \Gamma \vdash e_2 : \tau_2 \\
T_1 \tau_1 \sim U \tau_2 \rightarrow \alpha \\
\]

\[
UT_1 T \Gamma \vdash \text{Apply } e_1 e_2 : U \alpha \\
\]

\[
\text{Fst: } \forall a. \forall b. (a * b) \rightarrow a \in \Gamma \\
\vdash \text{Fst: } (x * y) \rightarrow x \\
\vdash (1, \text{True}): (\text{Int, Bool}) \\
\vdash \text{Apply Fst } (1, \text{True}): \text{Int} \\
\]

where \( U = \{ \alpha := \text{Int}, y := \text{Bool}, x := \text{Int} \} \)
Type Inference

- Simple function example:

\[
T(\Gamma \cup \{ x : \alpha_1 \} \cup \{ f : \alpha_2 \}) \vdash e : \tau \quad \alpha_2 \Upsilon T \alpha_1 \rightarrow \tau
\]

\[
UT\Gamma \vdash \text{Letfun}(f.x.e) : T \alpha_1 \rightarrow \tau
\]

\[
\text{[]}(\Gamma \cup \{ x : \alpha_1 \} \cup \{ f : \alpha_2 \}) \vdash (x,x) : (\alpha_1, \alpha_1) \quad \alpha_2 \sim [\alpha_1 \rightarrow (\alpha_1, \alpha_1)]
\]

\[
[\alpha_2 := \alpha_1 \rightarrow (\alpha_1, \alpha_1)] \Gamma \vdash \text{Letfun}(f.x.(x,x)) : \alpha_1 \rightarrow (\alpha_1, \alpha_1)
\]

\[\alpha_1, \alpha_2 \text{ fresh}\]
Type Inference

• Simple recursive function

\[
T(\Gamma \cup \{ x : \alpha_1 \} \cup \{ f : \alpha_2 \}) \vdash e : \tau \quad \alpha_2 \overset{U}{\sim} T \alpha_1 \rightarrow \tau
\]

\[
UT\Gamma \vdash \text{Letfun}(f.x.e) : T \alpha_1 \rightarrow \tau
\]

\[
\vdots
\]

\[
[\alpha_1 \rightarrow \alpha_3/\alpha_2](\Gamma \cup \{ x : \alpha_1 \} \cup \{ f : \alpha_2 \}) \vdash \text{App } f x : \alpha_3 \quad \alpha_2 \sim [\alpha_1 \rightarrow \alpha_3/\alpha_2](\alpha_1 \rightarrow \alpha_3)
\]

\[
[\alpha_1 \rightarrow \alpha_3/\alpha_2][\alpha_1 \rightarrow \alpha_3/\alpha_2] \Gamma \vdash \text{Letfun}(f.x.(\text{App } f x)) : \alpha_1 \rightarrow \alpha_3
\]
Re-introducing the ∀-quantor

- None of the rules so far re-introduced the ∀-quantor

- Is this necessary at all?

```plaintext
let
  f = letfun g x = (x,x)
  in (f True, f 1)
let
  f x = (x,x)
  in (f True, f 1)
```

- only necessary if we have let-bindings so polymorphic functions can be ‘exported’
Re-introducing the $\forall$-quantor

• Generalise over all variables which occur free in $\tau$, but not in $\Gamma$

  $\forall (TV(\tau) \setminus TV(\Gamma)) \cdot \tau$

★Example:

$Gen (\{x : a, y : Int\}, (a,b) \to b) = \forall b. (a,b) \to b$

\[
\begin{array}{c}
T_1 \Gamma \vdash e_1 : \tau \\
T_2 (T_1 \Gamma \cup x : Gen (T_1 \Gamma, \tau)) \vdash e_2 : \tau' \\
\hline
T_1 T_2 \Gamma \vdash (\text{Let } e_1 \ x.e_2) : \tau'
\end{array}
\]
Type Inference

• Rules for Plus, Mult, Inl, and Inr, can be derived from their type in and the application rule

• The inference rules describe Robin Milner’s type inference algorithm \( W \)

• Returns the same typing scheme as the non-syntax directed rules discussed previously (modulo \( \forall \)-quantification)
Unification

• Simple unification algorithm

  ▶ input: two type terms $t_1$ and $t_2$, $\forall$-quantified variables replaced by fresh, unique variables

  ▶ output: the most general unifier of $t_1$ and $t_2$ (if it exists)
Unification

• Cases \( t_1 \) and \( t_2 \)
  ★ are both type variables \( v_1 \) and \( v_2 \)
    ‣ if \( v_1 = v_2 \), return empty substitution
    ‣ otherwise return \([v_1/v_2]\)
  ★ are both primitive types
    ‣ if they are the same, return the empty substitution
    ‣ otherwise, there is no unifier
  ★ both are product types with \( t_1 = (t_{11}*t_{12}) \) and \( t_2 = (t_{21}*t_{22}) \)
    ‣ compute the mgu \( S \) of \( t_{11} \) and \( t_{21} \)
    ‣ compute the mgu \( S' \) of \( S t_{12} \) and \( S t_{22} \)
    ‣ return \( S \cup S' \)
  ★ both function types, sum types (see product types)
  ★ only one is is type variable \( v \), the other an arbitrary term \( t \)
    ‣ if \( v \) occurs in \( t \), there is no unifier (occurs check)
    ‣ otherwise, return \{\( v := t \}\)
  ★ otherwise, there is no unifier