Utrecht University

Concepts of Programming Language Design

Parametric Polymorphism

Gabriele Keller
Tom Smeding

Overview

higher & first-order syntax

inference rules, induction

tools to talk about languages

abstract machines

big step and small step operational semantics

value & type environments

parametric polymorphism/ generics

control stacks

partial application/function closures

semantic features

type checking

functional

(algebraic) data types

static & dynamic scoping

static & dynamic typing

language concepts

explicit & implicit typing

procedural/imperative

Utrecht University

Parametric Polymorphism

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

```
swap (x, y) = (y, x)
```

- What is `swap`'s type?
  - In Haskell: forall a. forall b. (a,b) -> (b,a)

```
swap :: (a, b) -> (b, a)
```

  - in MinHs:

```
recfun swapIntBool :: (Int, Bool) -> (Bool, Int) pair =
   (snd pair, fst pair)

recfun swapBoolInt :: (Bool, Int) -> (Int, Bool) pair =
   (snd pair, fst pair)
```

.....

Parametric polymorphism

The term polymorphism is used in the PL context to mean several different concepts:

- Parametric polymorphism (when functional programmers talk about polymorphism, often referred to as ‘generics’ by OO ppl)
  - the operation can work on any type

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)

swap (1,"Hello")

swap ('c', \x -> x + 1)

swap (True, odd)
```

Parametric polymorphism

- Adhoc polymorphism (when OO programmers talk about polymorphism)
  - a function or operation is overloaded with multiple implementations to work on some specific types

```
5 + 3

3.7 + 1.234

‘‘Hello’’ + ‘‘ World’’
```

Sometimes polymorphism is used to refer to subtyping

We will cover all of these concepts in the course

Parametric Polymorphism in Haskell

- Parametric polymorphism:

```
swap :: (a, b) -> (b, a)
swap 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:

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

  - instantiates type variable `a` to `Int`, `b` to `Bool`

Parametric Polymorphism (generics) in C#

```
static void Swap<T>(ref T a, ref T b) {
  T temp;
  temp = a;
  a    = b;
  b    = temp;
}
```

explicit introduction of type variable

```
static void Main(string[] args) {
  int a, b;
  char c, d;
  a = 5;
  b = 10;
  c = 'X';
  d = ‘Y';

  Swap<int>(ref a, ref b);
  Swap<char>(ref c, ref d);

 }
}
```

explicit instantiation of type variable

Parametric Polymorphism (generics) in Swift

```
func swap<T>(_ a: inout T, _ b: inout T) {
  let tmp = a
      a = b
      b = tmp
}

var x = 3
var y = 107

swap(&x, &y)

var str1 = ‘‘hello"
var str2 = "world"
swap(&str1, &str2)
```

explicit introduction of type variable

but no explicit instantiation necessary
(but possible)

Adding Parametric Polymorphism to MinHs

- First: with explicit typing (as in C#)
  - introduction of type variables is explicit

```
static void Swap<T>(ref T a, ref T b)
func swap<T>(_ a: inout T, _ b: inout T)
```

  - instantiation of type variables is explicit

```
Swap<int>(ref a, ref b)
```

- Later, we look into how this works for implicit typing (i.e., programmer does not have to provide the type

Adding Parametric Polymorphism to MinHs

- Type abstraction in polymorphic (explicitly typed) MinHs:

explicit introduction of type variables

```
(Type a in
  (Type b in
    recfun swap :: ((a * b) → (b * a)) pair =
      (snd pair, fst pair)))
```

- Type instantiation

explicit instantiation of type variables
(corresponds to application on the value level)

```
(inst (Type a in
  (inst (Type b in
     recfun swap :: ((a * b) → (b * a)) pair =
       (snd pair, fst pair)
     Bool)
   Int)))
```

evaluates to

```
recfun swap :: (Int * Bool) → (Bool * Int) pair =
  (snd pair, fst pair)
```

Utrecht University

Parametric Polymorphism

- What is the type of this function?

```
(Type a in
  (Type b in
    recfun swap :: ((a * b) → (b * a)) pair =
      (snd pair, fst pair)))
```

- Universal quantification:
  - it is $\forall a. \forall b. (a * b) \to (b * a)$
  - written in Haskell (leading `forall a. forall b.` optional)

```
forall a. forall b. (a * b) → (b * a)
```

Polymorphic MinHS - Concrete Syntax

Polytypes	σ	::=	τ | $\forall$ Ident . σ
Monotypes	τ	::=	`Bool` | `Int` | (τ → τ) | Ident
Expressions	Expr	::=	Ident
			| `inst` (Expr, τ)
			| `recfun` Ident `::` (τ → τ) Ident = Expr
			| `Type` Ident `in` Expr | ...

- Note that we only allow quantifiers at the outermost position
  - this restriction is not necessary for explicitly typed MinHs

Polymorphic MinHS

- Valid Types:
  - types can now contain type variables, but they need to be “in scope” (bound by a quantifier)

$$\frac{}{\Delta \vdash \texttt{Bool}\ \ \textit{ok}} \qquad \frac{}{\Delta \vdash \texttt{Int}\ \ \textit{ok}} \qquad \frac{\Delta \vdash \tau_1\ \textit{ok} \qquad \Delta \vdash \tau_1\ \textit{ok}}{\Delta \vdash \tau_1 \to \tau_2\ \textit{ok}}$$

$$\frac{\Delta \cup \{t\} \vdash \sigma\ \textit{ok} \qquad t \notin \Delta}{\Delta \vdash \forall\, t.\ \sigma\ \textit{ok}} \qquad \frac{t \in \Delta}{\Delta \vdash t\ \textit{ok}}$$

Polymorphic MinHS

- Typing rules

$$\frac{\Delta \cup \{t\}, \Gamma \vdash e : \sigma \qquad t \notin \Delta}{\Delta, \Gamma \vdash (\texttt{Type}\ (t.\,e)) : \forall t.\, \sigma}$$

$$\frac{\Delta, \Gamma \vdash e : \forall t.\, \sigma \qquad \Delta \vdash \boldsymbol{\tau}\ \boldsymbol{ok}}{\Delta, \Gamma \vdash (\texttt{Inst}\ e\ \boldsymbol{\tau}) : \sigma[t := \boldsymbol{\tau}]}$$

Utrecht University

Polymorphic MinHS

- Dynamic Semantics

$$\frac{e \mapsto_M e'}{(\texttt{Inst}\ \ e\ \ \tau) \mapsto_M (\texttt{Inst}\ \ e'\ \ \tau)}$$

$$\frac{}{(\texttt{Inst}(\texttt{Type}(t.e))\ \tau) \mapsto_M e[t := \tau]}$$

Polymorphic MinHs

- Polymorphic MinHs with
  - explicit introduction of type variables:
    - `(Type a in recfun id :: (a -> a) x = x)` :: ∀ a.a → a
  - explicit instantiation of type variables:
    - `(Inst(Type a in recfun id : (a->a) x = x) Bool)` :: Bool → Bool

Parametric Polymorphism

- We only allow quantifiers in a type at the outermost position. Does this matter?
  - Example: can we give a type to this function?

```
strangeFun f = if (f True)
               then (f 5)
               else 10
```

  - Possible type:

```
strangeFun :: (∀a. a → a) → Int
```

    but there is not possible type which would be legal in our MinHs definition!

- Polymorphic types are not first class citizens in MinHs!
  - we can’t specify that a function requires or produces a polymorphic function/value!

Implementing parametric polymorphism

- We discussed the implications of parametric polymorphism/generics for the static and dynamic semantics
- But how can it actually be implemented? What code should be generated by the compiler for polymorphic functions
- Let’s look how generics can be simulated in a language which doesn’t support parametric polymorphism

Implementing parametric polymorphism

C:

```
#define SWAP(x, y, T) {T SWAPTMP = x; x = y; y = SWAPTMP;}


double x = 10.12;
double y = 2.123;
SWAP(x, y, double);


void swap (void **x, void **y){
  void *tmp = *x;
  *x = *y;
  *y = *x;
}

double *a = malloc (sizeof (double));
*a = 10.12;

double *b = malloc (sizeof (double));
*b = 3.14;

swap((void**)&a, (void**)&b);
```

Implementing parametric polymorphism

- There are two choices (somewhat simplified):
  - generate monomorphic code for every instance used
    - code size?
    - separate compilation?
  - `box' every value, so all polymorphic operations can be expressed in terms on operations on pointers
    - runtime overhead?
    - locality?
  - monomorphisation optimisation: boxed values, but specialise to monomorphic version where possible
- OO languages where everything is an object and associated with a vTable have other options available (similar to overloading, will be covered later)

Polymorphism in implicitly typed languages

- Explicitly typed languages require type annotations for every new variable/function/method name
  - explicitly typed languages: C, C++, Fortran, Java, Objective C, Visual Basic, C# (2.0 and earlier)
- Implicitly typed languages make the compiler infer the correct type
  - implicitly typed languages: C# (>= 3.0), Haskell, Python, Go
- Some languages allow omission of types in some circumstances (Swift)
- Many implicitly typed languages allow the user to add optional type annotations
  - correctness
  - performance

From types to programs

- We showed that

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

has the type

```
∀a.∀b.(a * b) → (b * a)
```

Can we go backwards?

given a type, what is the program?

which other programs have this type?

From types to programs

- Observations:
  - a function can't do anything with a value of polymorphic type but to pass it along, copy it, or ignore it

  - Assume a function has this type:

```
f :: ∀a. [a] → Int
```

the `Int` value cannot depend on the values in the list!

it can only depend on the structure of the list!

  - Therefore, we know that for any function `g` and list `xs`

```
f (map g xs)   = f xs
```

From types to programs

- Observations:
  - only `Error` values can have type

$$\forall a.\ a$$

Total polymorphic functions

- For which of the following types can we write total, terminating MinHs functions?
  - $\forall a. \forall b.(a * b) \to (b * a)$ ✔
  - $\forall a. \forall b.(a + b) \to (b + a)$ ✔
  - $\forall a. \forall b.(a * b) \to a$ ✔
  - $\forall a. \forall b.(a + b) \to a$ ✘
  - $\forall a. \forall b.\ a \to (a + b)$ ✔
  - $\forall a. \forall b.(a \to b) \to (b \to a)$ ✘
  - $\forall a. \forall b. \forall c. ((a \to c) * (b \to c)) \to (a+b \to c)$ ✔

Total polymorphic functions

- Curry-Howard correspondence:
  - The type constructors `+`, `*` , and $\rightarrow$ correspond to the logical operators $\vee$,$\wedge$ and $\Rightarrow$
  - Types correspond to theorems, and total, terminating programs to (constructive) proofs
  - A type checker then corresponds to a proof checker!
  - our MinHs types correspond to propositional formulae, therefore the theorems are not very exciting
  - more powerful type systems correspond to more powerful logics

Total polymorphic functions

- Some propositional logic theorems:

$$(A \wedge B) \Rightarrow A$$

$$(A \wedge B) \Rightarrow B$$

$$A \Rightarrow (A \vee B)$$

$$B \Rightarrow (A \vee B)$$

$$(A \wedge (A \Rightarrow B)) \Rightarrow B$$

- Currying /uncurrying

$$a \rightarrow (b \rightarrow c) \quad \equiv \quad (a * b) \rightarrow c$$

$$A \Rightarrow (B \Rightarrow C) \quad \equiv \quad (A \wedge B) \Rightarrow C$$

- What does the unit type correspond to?
- What about primitive types (int, bool)?

Utrecht University

Concepts of Programming Language Design

Parametric Polymorphism: Type Inference

Gabriele Keller
Tom Smeding

Polymorphism

- Parametric polymorphism:
  - we already know how to type check an explicitly typed polymorphic program
  - today we discuss how to infer the type of a polymorphic program
- Looking at other flavours of polymorphism:
  - Subtyping
  - Subclassing in the context of OO (Featherweight Java), overriding
  - Overloading

How can a compiler infer types?

Principal Type

- What type should the compiler infer for function `f`?

```
recfun f x = (fst x) + 1
```

- Possible types

  (1) `Int* Int ➔ Int`

  (2) `Int* Bool ➔ Int`

  (3) `Int* (Int -> (Int + Bool)) ➔ Int`

  (4) `∀ a. Int * a ➔ Int`

- Types (1) - (3) are instances of type (4)

Principal Type

- We write $\tau' \leq \tau$ if τ' is less general than τ, or in other words, is τ' an instance of τ

  - `Int * Int → Int` $\leq$ `∀ a. Int * a → Int`
  - `∀ a. Int * a → Int` $\leq$ `∀ a. ∀ b. b * a → b`
  - `∀ a. Int * a → Int` $\not\leq$ `∀ a. a * a → a`
  - `∀ a. a * a → a` $\leq$ `∀ a. ∀ b. b * a → b`
  - `∀ a. (a * a) * (a* a) → (a* a)` $\leq$ `∀ a. a * a → a`

- More formally:

  - $\forall\, b_1 \ldots b_k.\ \tau' \leq \forall\, a_1 \ldots a_n.\ \tau$ if there is a substitution S such that $\tau' = (S\ \tau)$

- We are interested in the most general type τ 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
  - no explicit type abstraction and instantiation (`Type` and `Inst` not part of the language)
  - Types of the build-in functions and constructors are part of the environment:
    - $\Gamma = \{\texttt{+ : Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int},\ \texttt{fst : } \forall\, \texttt{a}.\ \forall\, \texttt{b}.\, \texttt{(a * b)} \rightarrow \texttt{a}, \ldots\}$
  - no overloading yet, e.g., `==` still only compares integers,

Implicitly Typed MinHs

- What is the type of the following expressions:
  - `Inl True`
    - we would not be able to determine the type in a monomorphic setting
    - polymorphic type: `∀ a. (Bool + a)`
  - `Fst (1, True)`
    - type of `Fst`: `∀ a. ∀ b.(a * b) → a`
    - type of argument `(Int * Bool)`
    - type: `Int`
  - `Roll (Inl 1))`
    - impossible to derive a most general type in implicitly typed language, therefore not part of the language (named recursive types are no problem!)

Typing Rules

- First, let us look at typing rules that are sufficient to derive

$$\Gamma \vdash e : \sigma$$

if σ is a possible (possibly polymorphic) type of e under the environment Γ

Typing Rules

- Can we use the type checking rules to infer the type?
- Application, if-expression, variable and product rules:

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

input: env Γ and expression

output: type of expression τ

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (\texttt{Pair}\ t_1\ t_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (\texttt{Apply}\ t_1\ t_2) : \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash (\texttt{If}\ t_1\ t_2\ t_3) : \tau}$$

Typing Rules

- Functions, `Inr`, `Inl`

problem: env Γ is not an input!

we have to guess the types of f and x

$$\frac{\Gamma \cup \{f : \tau_1 \rightarrow \tau_2\ ,\ x : \tau_1\} \vdash\ t : \tau_2}{\Gamma\ \vdash (\texttt{Recfun}\ \ (f.x.t)) : \tau_1 \rightarrow \tau_2}$$

problem: type is not an output,

we have to guess τ_2

$$\frac{\Gamma \vdash\ t_1 : \tau_1}{\Gamma \vdash\ (\texttt{Inl}\ \ t_1) :\ \tau_1 + \tau_2}$$

Typing Rules

- ∀ -introduction and elimination

problem: type not an output (guess)

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

$$\frac{\Gamma \vdash e : \tau \qquad t \notin \mathit{FreeTypeVars}(\Gamma)}{\Gamma \vdash e : \forall\, t.\tau}$$

problem: not syntax directed

Type Inference Algorithm - summary so far

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

$$\frac{\Gamma \vdash e : \tau \qquad t \notin \mathit{FreeTypeVars}(\Gamma)}{\Gamma \vdash e : \forall\, t.\tau}$$

$$\frac{\Gamma \cup \{f : \tau_1 \to \tau_2\,,\ x : \tau_1\} \vdash t : \tau_2}{\Gamma \vdash (\texttt{Recfun}\ \ (f.x.t)) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash (\texttt{Inl}\ t_1) : \tau_1 + \tau_2}$$

- The rules above do **not** specify a type inference algorithm:
  - it is not possible to view the environment and the expression as input, the type as an output
  - the rules are not syntax directed

Type Inference Algorithm

- Idea
  - delay the instantiation of type variables until necessary
  - replace ∀-quantified variables by free, fresh variables

Type inference algorithm

- The type of an expression alone as output is not sufficient:

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (\texttt{If}\ t_1\ t_2\ t_3) : \tau}$$

```
f x y  = if True       assume x has unknown type a, y unknown type b
            then (x, y + 1)      (a    * Int)         a must be Bool
            else (False, y)      (Bool *    b)        b must be Int
```

- Inspecting the then-branch reveals that it has type pair of something and integer, but also that `y` has to have type `Int`

- Since both branches have to have the same type, we know that `x` has to have type `Bool`

- By looking at the if-expression, we can determine that is has type `(Bool * Int)`, but also what type variables `a` and `b` are standing for

Type Inference Algorithm

- Idea
  - delay the instantiation of type variables until necessary
  - replace ∀-quantified variables by free, fresh variables
  - find a substitution to unify the derived with the required type
  - make the substitution part of the result of the type inference

- Input: expression e and environment Γ
- Output: type of expression τ, substitution S with possible instantiations of type variables in

$$S\Gamma \vdash e : \tau$$

```
[a := Bool, b := Int]] {x :: a, y :: b} ⊢ (If True (Pair x (Plus y 1)) (Pair False y)):: (Bool * Int)
```

Type Inference Algorithm

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

$$\texttt{(Bool * x)[y := Bool,x := Int]} \stackrel{?}{=} \texttt{(y * Int) [y :=Bool,x := Int]}$$

  or to replace variables with other variables

What about

$$\texttt{(x * x) [x:=y]} = \texttt{(x * y) [x:=y]}$$

$$\texttt{(x * x)[x:=Int, y := Int]} = \texttt{(x * y)[x:=Int, y := Int]}$$

Unification

- A substitution S, with $S\ \tau = S\ \tau'$ is called a unifier of τ and τ'
- For the algorithm, we need the most general unifier (mgu)
  - there may be more than one mgu
  - resulting terms are the same module renaming
- We write $\tau_1 \overset{S}{\sim} \tau_2$ if S is an mgu of τ_1 and τ_2
- Examples:
  - are there mgu's for the following pairs of types?

$$\texttt{(a *(a * a))} \overset{?}{=} \texttt{(b * c)}$$

$$\texttt{Int} \overset{?}{=} \texttt{Bool}$$

$$\texttt{a} \overset{?}{=} \texttt{(a * a)}$$

Unification

- Simple unification algorithm
  - input: two type terms t_1 and t_2, ∀-quantified variables replaced by fresh, unique variables
  - output: the most general unifier of t_1 and t_2 (if it exists)

Unification - Computing the Most General Unifier (MGU)

- 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 $[v_2 := v_1]$
  - 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 $[v := t]$

Unification - Computing the Most General Unifier (MGU)

- We discussed how to calculate the Most General Unifier of two type terms:
  - most general substitution to unify two type terms:

```
f x y  = if True
         then (x, y + 1) (a    * Int)
         else (False, y) (Bool * b)
```

$$\texttt{(Bool * b)} \quad \overset{\texttt{[a := Bool, b := Int]}}{\sim} \quad \texttt{(a * Int)}$$

Type Inference Algorithm

- Now back to our type inference algorithm
  - $T\,\Gamma \vdash e : \tau$
  - ∀- elimination:

β_i fresh

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

compare to previous rule:

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

Type Inference Algorithm

- If-rule:

$$\frac{T\ \Gamma \vdash e_c : \tau_c \qquad \tau_c \overset{U}{\sim} \texttt{Bool} \qquad T_t U T\ \Gamma \vdash e_t : \tau_t \qquad T_e T_t U T \Gamma \vdash e_e : \tau_e \qquad T_e \tau_t \overset{U'}{\sim} \tau_e}{U' U T T_t T_e\ \Gamma \vdash (\texttt{If}\ e_c\ e_t\ e_e) : U'\ \tau_e}$$

- algorithmic interpretation:
  - Input : Γ and expression $(\texttt{If}\ e_c\ e_t\ e_e)$
  - first, derive type of expression e_c with environment Γ
    - result: the substitution T and the type τ_c
  - unify the types τ_c and `Bool`
    - result: substitution U
  - derive type of expression e_t with new environment $UT\Gamma$
    - result: the substitution T_t and the type τ_t
  - derive type of expression e_e with new environment $T_t U T\ \Gamma$
    - result: the substitution T_e and the type τ_e
  - unify the types $T_t\ \tau_t$ and τ_e
    - result: substitution U'
  - return substitution $U' U T T_t T_e$ and type $U'\ \tau_e$

Type Inference Algorithm

- Application rule

$$\frac{T\,\Gamma \vdash e_1 : \tau_1 \qquad T_1 T\,\Gamma \vdash e_2 : \tau_2 \qquad T_1\ \tau_1 \overset{U}{\sim} \tau_2 \to \alpha}{U T_1 T\,\Gamma \vdash (\texttt{Apply}\ e_1\ e_2) : U\alpha}$$

α fresh

- algorithmic interpretation:
  - Input : Γ and expression $(\texttt{Apply}\ e_1\ e_2)$
  - first, derive type of expression e_1 with environment Γ
    - result: the substitution T and the type τ_1
  - now, derive type of expression e_2 with new environment $T\,\Gamma$
    - result: the substitution T_1 and the type τ_2
  - now, unify the types $T_1\ \tau_1$ and $\tau_2 \to \alpha$
    - result: substitution U
  - return substitution $U T_1 T$ and type $U\alpha$

Type Inference Algorithm

- Note
  - the rules are syntax directed
    - for every type of expression, there is exactly one rule which applies
  - the environment and expression are input & unifier and type are output

Type Inference Algorithm

- Function rule

$$\frac{T(\Gamma \cup \{x : a_1\} \cup \{f : a_2\}) \vdash e : \tau \qquad T\ a_2 \overset{U}{\sim} T\ a_1 \rightarrow \tau}{UT\Gamma \vdash (\texttt{Recfun}\ (f.x.e)) : U\ (T\ a_1 \rightarrow \tau)} \qquad a_i \text{ fresh}$$

Re-introducing the ∀-quantifier

- None of the rules so far re-introduced the ∀-quantifier
- Is this necessary at all?

```
let
  f = recfun 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 (or global function bindings) so polymorphic functions can be applied in different contexts

Re-introducing the ∀-quantifier

- Generalise over all variables which occur free in τ, but not in Γ
  - Let $\mathbf{TV}(\Gamma)$ be the set of all free type variables in Γ, $\mathbf{TV}(\tau)$ the set of all free type variables in τ
  - Define $\mathbf{Gen}(\Gamma, \tau)$ as
    - $\mathbf{Gen}(\Gamma, \tau) = \forall(\mathbf{TV}(\tau) \setminus \mathbf{TV}(\Gamma)).\ \tau$

- Example:
  - $\mathbf{Gen}(\{x : a, y: \texttt{Int}\}, (a*b) \to b) = \forall b.\ (a*b) \to b$

- New ∀-introduction rule:

$$\frac{T_1\ \Gamma \vdash e_1 : \tau \qquad T_2\ (T_1\Gamma \cup x : \mathbf{Gen}(T_1\Gamma, \tau)) \vdash e_2 : \tau'}{T_1\ T_2\ \Gamma \vdash (\texttt{Let}\ e_1\ (x.\ e_2)) : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \qquad t \notin \mathit{FreeTypeVars}(\Gamma)}{\Gamma \vdash e : \forall\, t.\tau}$$

Example

$$\frac{T\,\Gamma \vdash e_1 : \tau_1 \qquad T_1 T\,\Gamma \vdash e_2 : \tau_2 \qquad T_1\,\tau_1 \overset{U}{\sim} \tau_2 \to \alpha}{U T_1 T\,\Gamma \vdash \texttt{(Apply}\ e_1\ e_2\texttt{)} : U\alpha}$$

α fresh

`(Apply Fst (Pair 1 True))`