

# Programming Languages, CPSC 604—Slides 12

## Polymorphic Lambda Calculus

Jaakko Järvi

April 2, 2009

# Outline

- 1 Introduction
- 2 Aside: general recursion
- 3 About System F
- 4 Existential types
- 5 Abstract data types
- 6 Existential objects

# Polymorphism

- Simply typed lambda-calculus is quite frustrating to program with (even with our fancy extensions thus far)
- Task: write the identity function

# Polymorphism

- Simply typed lambda-calculus is quite frustrating to program with (even with our fancy extensions thus far)
- Task: write the identity function

$\lambda x : \text{bool}.x$

$\lambda x : \text{nat}.x$

$\lambda x : \text{bool}.x$

$\lambda x : \text{bool}.x$

$\lambda x : \text{bool} \rightarrow \text{nat}.x$

...

# Polymorphism

- Simply typed lambda-calculus is quite frustrating to program with (even with our fancy extensions thus far)
- Task: write the identity function

$\lambda x : \text{bool}.x$

$\lambda x : \text{nat}.x$

$\lambda x : \text{bool}.x$

$\lambda x : \text{bool}.x$

$\lambda x : \text{bool} \rightarrow \text{nat}.x$

...

- That's almost like Pascal with fixed-size arrays!
  - `sort2elements`, `sort3elements`, ...

# Polymorphism

- Consider a sorting function (assuming we have defined a type constructor `list`)
- Here's one possible type for it:

```
sort : list nat → (nat → nat → bool) → list nat
```

- Here is another one:

```
sort : list {nat, nat} →  
      ({nat, nat} → {nat, nat} → bool) →  
      list {nat, nat}
```

- How do their implementations differ?

# Polymorphism

- Consider a sorting function (assuming we have defined a type constructor `list`)
- Here's one possible type for it:

```
sort : list nat → (nat → nat → bool) → list nat
```

- Here is another one:

```
sort : list {nat, nat} →  
      ({nat, nat} → {nat, nat} → bool) →  
      list {nat, nat}
```

- How do their implementations differ?
- They don't. Two implementations are only necessary to appease the type checker

## Polymorphic types (mainly functions)

- A polymorphic function accepts many types of arguments
- Commonly different forms of polymorphism are divided as:
  - subtype polymorphism (bounded polymorphism)
  - ad-hoc polymorphism (C++ function overloading)
  - parametric polymorphism (same function works for all types)
    - also generics, genericity, templates, ...
- A polymorphic function is parameterized over types

## Polymorphic types (mainly functions)

- A polymorphic function accepts many types of arguments
- Commonly different forms of polymorphism are divided as:
  - subtype polymorphism (bounded polymorphism)
  - ad-hoc polymorphism (C++ function overloading)
  - **parametric polymorphism** (same function works for all types)
    - also generics, genericity, templates, ...
- A polymorphic function is parameterized over types

# System F

- Girard (1972), John Reynolds (1974)
- Also, “Second-order lambda-calculus”
- Basically, extends simply-typed lambda calculus with the ability to write lambda abstractions over types, and perform substitution with types

## System-F on one page

## • Syntax

$$t ::= x \mid v \mid t t \mid t[T]$$

$$v ::= \lambda x : T. t \mid \Lambda X. t$$

$$T ::= X \mid T \rightarrow T \mid \forall X. T$$

## • Evaluation rules

E-AppFun

$$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}$$

E-AppArg

$$\frac{t \rightarrow t'}{v t \rightarrow v t'}$$

E-AppAbs

$$(\lambda x : T. t) v \rightarrow [v/x]t$$

E-TypeApp

$$\frac{t_1 \rightarrow t_1'}{t_1[T] \rightarrow t_1'[T]}$$

E-TypeAppAbs

$$(\Lambda X. t)[T] \rightarrow [T/X]t$$

## • Typing rules

T-Variable

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

T-Abstraction

$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T. u : T \rightarrow U}$$

T-Application

$$\frac{\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U}{\Gamma \vdash t u : T}$$

T-TypeAbstraction

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X. t : \forall X. T}$$

T-TypeApplication

$$\frac{\Gamma \vdash t : \forall X. T}{\Gamma \vdash t[T_1] : [T_1/X]T}$$

# New rules

- Evaluation

$$\frac{\text{E-TypeApp} \quad t_1 \rightarrow t_1'}{t_1[T] \rightarrow t_1'[T]} \quad \text{E-TypeAppAbs} \quad (\lambda X.t)[T] \rightarrow [T/X]t$$

- Typing

$$\frac{\text{T-TypeAbstraction} \quad \Gamma, X \vdash t : T}{\Gamma \vdash \lambda X.t : \forall X.T} \quad \text{T-TypeApplication} \quad \frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t[T_1] : [T_1/X]T}$$

- Note the type variables in the environment
  - For now, just type variables, no binding to anything
  - Same conventions: must do renaming if a type variable name already bound

# Examples

- What are the types of these?

$id = \Lambda T. \lambda x : T. x$

$id[\text{bool}]$

$id[\text{bool}] \text{ true}$

$id \text{ true}$

# Examples

- What are the types of these?

$id = \Lambda T. \lambda x : T. x \quad : \forall X. X \rightarrow X$

$id[\text{bool}]$

$id[\text{bool}] \text{ true}$

$id \text{ true}$

# Examples

- What are the types of these?

$id = \Lambda T. \lambda x : T. x \quad : \forall X. X \rightarrow X$

$id[\text{bool}] \quad : \text{bool} \rightarrow \text{bool}$

$id[\text{bool}] \text{ true}$

$id \text{ true}$

# Examples

- What are the types of these?

$id = \Lambda T. \lambda x : T. x \quad : \forall X. X \rightarrow X$

$id[\text{bool}] \quad : \text{bool} \rightarrow \text{bool}$

$id[\text{bool}] \text{ true} \quad : \text{bool}$

$id \text{ true}$

# Examples

- What are the types of these?

$id = \Lambda T. \lambda x : T. x$     :  $\forall X. X \rightarrow X$

$id[\text{bool}]$                 :  $\text{bool} \rightarrow \text{bool}$

$id[\text{bool}] \text{ true}$          :  $\text{bool}$

$id \text{ true}$                  type error!

# Examples

- What are the types of these?

$id = \Lambda T. \lambda x : T. x \quad : \forall X. X \rightarrow X$

$id[\text{bool}] \quad : \text{bool} \rightarrow \text{bool}$

$id[\text{bool}] \text{ true} \quad : \text{bool}$

$id \text{ true} \quad \text{type error!}$

- Thus, no type argument deduction (also “implicit instantiation”)

Example derivation  $id$  [bool] true

---

$$\vdash ((\lambda T. \lambda x : T. x)[\text{bool}]) \text{ true} : \text{bool}$$

Example derivation  $id$  [bool] true $\vdash \text{true} : \text{bool}$ 

---

 $\vdash ((\lambda T. \lambda x : T. x)[\text{bool}]) \text{true} : \text{bool}$

# Example derivation $id$ [bool] true

$$\frac{}{\vdash (\lambda T. \lambda x : T. x) \text{ [bool]} : \text{bool} \rightarrow \text{bool}}$$

$$\vdash \text{true} : \text{bool} \quad \text{T-True}$$

$$\frac{}{\vdash ((\lambda T. \lambda x : T. x) \text{ [bool]}) \text{ true} : \text{bool}}$$

# Example derivation $id$ $[bool]$ $true$

$$\frac{\frac{\frac{}{\vdash \Lambda T. \lambda x : T. x : \forall T. T \rightarrow T}}{\vdash (\Lambda T. \lambda x : T. x) [bool] : bool \rightarrow bool} \text{ T-TypeApp.}}{\vdash ((\Lambda T. \lambda x : T. x)[bool]) true : bool} \text{ T-True}$$

Example derivation *id* [bool] true

$$\begin{array}{c}
 \frac{}{T \vdash \lambda x : T. x : T \rightarrow T} \\
 \frac{}{\vdash \Lambda T. \lambda x : T. x : \forall T. T \rightarrow T} \text{ T-TypeAbs.} \\
 \frac{}{\vdash (\Lambda T. \lambda x : T. x) [\text{bool}] : \text{bool} \rightarrow \text{bool}} \text{ T-TypeApp.} \quad \vdash \text{true} : \text{bool} \text{ T-True} \\
 \hline
 \vdash ((\Lambda T. \lambda x : T. x) [\text{bool}]) \text{true} : \text{bool}
 \end{array}$$

Example derivation  $id$  [bool] true

$$\begin{array}{c}
 \frac{x : T \in T, x : T}{T, x : T \vdash x : T} \text{T-Var.} \\
 \frac{}{T \vdash \lambda x : T. x : T \rightarrow T} \text{T-Abs.} \\
 \frac{}{\vdash \Lambda T. \lambda x : T. x : \forall T. T \rightarrow T} \text{T-TypeAbs.} \\
 \frac{}{\vdash (\Lambda T. \lambda x : T. x) [\text{bool}] : \text{bool} \rightarrow \text{bool}} \text{T-TypeApp.} \quad \vdash \text{true} : \text{bool} \text{ T-True} \\
 \hline
 \vdash ((\Lambda T. \lambda x : T. x) [\text{bool}]) \text{true} : \text{bool}
 \end{array}$$

# Example derivation $id$ [bool] true

$$\begin{array}{c}
 \frac{x : T \in T, x : T}{T, x : T \vdash x : T} \text{ T-Var.} \\
 \frac{}{T \vdash \lambda x : T. x : T \rightarrow T} \text{ T-Abs.} \\
 \frac{}{\vdash \Lambda T. \lambda x : T. x : \forall T. T \rightarrow T} \text{ T-TypeAbs.} \\
 \frac{}{\vdash (\Lambda T. \lambda x : T. x) [\text{bool}] : \text{bool} \rightarrow \text{bool}} \text{ T-TypeApp.} \quad \vdash \text{true} : \text{bool} \text{ T-True} \\
 \hline
 \vdash ((\Lambda T. \lambda x : T. x) [\text{bool}]) \text{true} : \text{bool}
 \end{array}$$

- Compare T-Application and T-TypeApplication rules. Why no requirement on the type argument in T-TypeApplication?

## More interesting examples

- The doubling function:

$$\text{double} = \Lambda T. \lambda f : T \rightarrow T. \lambda x : T. f (f x)$$

- Instantiated with `nat`

$$\begin{aligned} \text{double\_nat} &= \text{double } [\text{nat}] \\ &: (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{nat} \end{aligned}$$

- Instantiated with `nat → nat`

$$\begin{aligned} \text{double\_nat\_arrow\_nat} &= \text{double } [\text{nat} \rightarrow \text{nat}] \\ &: ((\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \\ &\rightarrow \text{nat} \end{aligned}$$

- Invoking `double`:

$$\text{double } [\text{nat}] (\lambda x : \text{nat}. \text{succ } (\text{succ } x)) 5$$

## More interesting examples

- The doubling function:

$$\text{double} = \Lambda T. \lambda f : T \rightarrow T. \lambda x : T. f (f x)$$

- Instantiated with `nat`

$$\begin{aligned} \text{double\_nat} &= \text{double } [\text{nat}] \\ &: (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{nat} \end{aligned}$$

- Instantiated with `nat → nat`

$$\begin{aligned} \text{double\_nat\_arrow\_nat} &= \text{double } [\text{nat} \rightarrow \text{nat}] \\ &: ((\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \\ &\rightarrow \text{nat} \end{aligned}$$

- Invoking `double`:

$$\text{double } [\text{nat}] (\lambda x : \text{nat}. \text{succ } (\text{succ } x)) 5 \rightarrow 9$$

## Quadrupling — by applying double to itself

- $\text{double} = \Lambda T. \lambda f : T \rightarrow T. \lambda x : T. f (f x)$
- Now, define a function `quadruple` that calls `double` twice
- Quadrupling function:
  - $\text{quadruple} = \Lambda T. \text{double } [T \rightarrow T] (\text{double } [T])$
  - $\text{quadruple} : \forall T. (T \rightarrow T) \rightarrow T \rightarrow T$
- `double` has two different instantiations (so two different types) within the definition of `quadruple`.
  - `quadruple` cannot be typed in simply typed lambda-calculus

# Quadrupling — example reduction

- Reduce `quadruple [nat] succ 5`

## Quadrupling — example reduction

- Reduce quadruple `[nat] succ 5`

`Λ T.double [T → T] (double [T]) [nat] succ 5`

## Quadrupling — example reduction

- Reduce quadruple [nat] succ 5

```
 $\Lambda T.$ double [T  $\rightarrow$  T] (double [T]) [nat] succ 5  
double [nat  $\rightarrow$  nat] (double [nat]) succ 5
```

# Quadrupling — example reduction

- Reduce `quadruple [nat] succ 5`

$\Lambda T.$ `double [T → T] (double [T]) [nat] succ 5`

`double [nat → nat] (double [nat]) succ 5`

$\Lambda T.$ `λf : T → T.`

`λa : T.f (f a) [nat → nat] (double [nat]) succ 5`

# Quadrupling — example reduction

- Reduce `quadruple [nat] succ 5`

$\Lambda T.$ `double [T → T] (double [T]) [nat] succ 5`

`double [nat → nat] (double [nat]) succ 5`

$\Lambda T.$ `λf : T → T.`

`λa : T.f (f a) [nat → nat] (double [nat]) succ 5`

`(λf : (nat → nat) → nat → nat.`

`λa : nat → nat.f (f a)) (double [nat]) succ 5`

# Quadrupling — example reduction

- Reduce quadruple [nat] succ 5

$\Lambda T.$ double [T → T] (double [T]) [nat] succ 5

double [nat → nat] (double [nat]) succ 5

$\Lambda T.$ λf : T → T.

λa : T.f (f a) [nat → nat] (double [nat]) succ 5

(λf : (nat → nat) → nat → nat.

λa : nat → nat.f (f a)) (double [nat]) succ 5

(λa : nat → nat.double [nat] (double [nat] a)) succ 5

# Quadrupling — example reduction

- Reduce quadruple [nat] succ 5

$\Lambda T.$ double [T → T] (double [T]) [nat] succ 5

double [nat → nat] (double [nat]) succ 5

$\Lambda T.$ λf : T → T.

λa : T.f (f a) [nat → nat] (double [nat]) succ 5

(λf : (nat → nat) → nat → nat.

λa : nat → nat.f (f a)) (double [nat]) succ 5

(λa : nat → nat.double [nat] (double [nat] a)) succ 5

double [nat] (double [nat] succ) 5

# Quadrupling — example reduction

- Reduce quadruple [nat] succ 5

$\Lambda T.$ double [T → T] (double [T]) [nat] succ 5

double [nat → nat] (double [nat]) succ 5

$\Lambda T.$ λf : T → T.

λa : T.f (f a) [nat → nat] (double [nat]) succ 5

(λf : (nat → nat) → nat → nat.

λa : nat → nat.f (f a)) (double [nat]) succ 5

(λa : nat → nat.double [nat] (double [nat] a)) succ 5

double [nat] (double [nat] succ) 5 double [nat] (λa : nat.succ (succ a)) 5

# Quadrupling — example reduction

- Reduce quadruple [nat] succ 5

$\Lambda T.$ double [T → T] (double [T]) [nat] succ 5

double [nat → nat] (double [nat]) succ 5

$\Lambda T.$ λf : T → T.

λa : T.f (f a) [nat → nat] (double [nat]) succ 5

(λf : (nat → nat) → nat → nat.

λa : nat → nat.f (f a)) (double [nat]) succ 5

(λa : nat → nat.double [nat] (double [nat] a)) succ 5

double [nat] (double [nat] succ) 5 double [nat] (λa : nat.succ (succ a)) 5

λa' : nat.(λa : nat.succ (succ a))(λa : nat.succ (succ a)) a') 5

# Quadrupling — example reduction

- Reduce quadruple `[nat] succ 5`

$\Lambda T.$ double  $[T \rightarrow T]$  (double  $[T]$ )  $[nat]$  succ 5

double  $[nat \rightarrow nat]$  (double  $[nat]$ ) succ 5

$\Lambda T.$  $\lambda f : T \rightarrow T.$

$\lambda a : T.f (f a)$   $[nat \rightarrow nat]$  (double  $[nat]$ ) succ 5

$(\lambda f : (nat \rightarrow nat) \rightarrow nat \rightarrow nat.$

$\lambda a : nat \rightarrow nat.f (f a))$  (double  $[nat]$ ) succ 5

$(\lambda a : nat \rightarrow nat.double [nat]$  (double  $[nat]$   $a))$  succ 5

double  $[nat]$  (double  $[nat]$  succ) 5 double  $[nat]$   $(\lambda a : nat.succ (succ a))$  5

$\lambda a' : nat.(\lambda a : nat.succ (succ a))(\lambda a : nat.succ (succ a)) a')$  5

$(\lambda a : nat.succ (succ a))(\lambda a : nat.succ (succ a)) 5)$

# Quadrupling — example reduction

- Reduce quadruple `[nat] succ 5`

$\Lambda T.$ double  $[T \rightarrow T]$  (double  $[T]$ )  $[nat]$  succ 5

double  $[nat \rightarrow nat]$  (double  $[nat]$ ) succ 5

$\Lambda T.$  $\lambda f : T \rightarrow T.$

$\lambda a : T.f (f a)$   $[nat \rightarrow nat]$  (double  $[nat]$ ) succ 5

$(\lambda f : (nat \rightarrow nat) \rightarrow nat \rightarrow nat.$

$\lambda a : nat \rightarrow nat.f (f a))$  (double  $[nat]$ ) succ 5

$(\lambda a : nat \rightarrow nat.double [nat]$  (double  $[nat]$   $a))$  succ 5

double  $[nat]$  (double  $[nat]$  succ) 5 double  $[nat]$   $(\lambda a : nat.succ (succ a))$  5

$\lambda a' : nat.(\lambda a : nat.succ (succ a))(\lambda a : nat.succ (succ a)) a')$  5

$(\lambda a : nat.succ (succ a))(\lambda a : nat.succ (succ a))$  5)

$(\lambda a : nat.succ (succ a))(succ (succ 5))$

# Quadrupling — example reduction

- Reduce quadruple `[nat] succ 5`

$\Lambda T.$ double  $[T \rightarrow T]$  (double  $[T]$ )  $[nat]$  succ 5

double  $[nat \rightarrow nat]$  (double  $[nat]$ ) succ 5

$\Lambda T.$  $\lambda f : T \rightarrow T.$

$\lambda a : T.f (f a)$   $[nat \rightarrow nat]$  (double  $[nat]$ ) succ 5

$(\lambda f : (nat \rightarrow nat) \rightarrow nat \rightarrow nat.$

$\lambda a : nat \rightarrow nat.f (f a))$  (double  $[nat]$ ) succ 5

$(\lambda a : nat \rightarrow nat.double [nat]$  (double  $[nat]$   $a))$  succ 5

double  $[nat]$  (double  $[nat]$  succ) 5 double  $[nat]$   $(\lambda a : nat.succ (succ a))$  5

$\lambda a' : nat.(\lambda a : nat.succ (succ a))(\lambda a : nat.succ (succ a)) a')$  5

$(\lambda a : nat.succ (succ a))(\lambda a : nat.succ (succ a))$  5)

$(\lambda a : nat.succ (succ a))(succ (succ 5))$

succ (succ (succ (succ 5)))

# Self application

- In general, self application is not typable in simply typed lambda-calculus

$$\lambda x : ?.x \ x$$

- In system-F:

$$\text{selfapp} = \lambda x : \forall T. T \rightarrow T. x \ [\forall T. T \rightarrow T] \ x$$

$$\text{selfapp} : (\forall T. T \rightarrow T) \rightarrow (\forall T. T \rightarrow T)$$

# Outline

- 1 Introduction
- 2 Aside: general recursion**
- 3 About System F
- 4 Existential types
- 5 Abstract data types
- 6 Existential objects

# The fix operator

- We discussed the fix point operator (Y-combinator, `fix`), and showed its definition in untyped lambda calculus
- Just like self-application, `fix` cannot be typed in simply-typed lambda calculus
- Simple fix: add `fix` as a primitive

$$\text{fix } (\lambda x : T.t) \rightarrow [(\text{fix } (\lambda x : T.t))/x] t$$

$$\frac{t \rightarrow t'}{\text{fix } t \rightarrow \text{fix } t'}$$

$$\frac{\Gamma \vdash t : T \rightarrow T}{\Gamma \vdash \text{fix } t : T}$$

## Example

```
f = λ e:nat → bool.
  λ x:nat.
    if iszero x then true
    else
      if iszero (pred x) then false
      else e (pred (pred x))
```

```
f : (nat → bool) → nat → bool
```

```
iseven = fix f
```

```
iseven : nat → bool
```

- `f` is a generator for the `iseven` function
  - Given a function that **approximates** `iseven` for numbers up to  $n$ , `f` will define an approximation up to  $n + 2$
  - `fix f` extends this to all  $n$

# Lists

- New type: `List T`
- New syntax

```

nil[T]
cons[T] t t
isnil[T] t
head[T] t
tail[T] t

```

- Congruence rules...

$$\frac{t_1 \rightarrow t'_1}{\text{cons}[T] t_1 t_2 \rightarrow \text{cons}[T] t'_1 t_2}$$

- do the rest at home...

## Evaluating and typing lists

- Computation rules, just one example here, rest at home...

$$\text{head}[S] (\text{cons}[T] v_1 v_2) \rightarrow v_1$$

- Typing rules. Again, one example, rest at home....

$$\frac{\Gamma \vdash t : \text{List } T}{\Gamma \vdash \text{head}[T] t : T}$$

## Evaluating and typing lists

- Computation rules, just one example here, rest at home...

$$\text{head}[S] (\text{cons}[T] v_1 v_2) \rightarrow v_1$$

- Typing rules. Again, one example, rest at home...

$$\frac{\Gamma \vdash t : \text{List } T}{\Gamma \vdash \text{head}[T] t : T}$$

- Both `fix` and lists were **extensions to simply typed lambda calculus**, with their own special typing and evaluation rules

## Back to system-F

- System-F is a more expressive formalism
  - Recursive structures, as well as polymorphic types (such as lists) can be typed
- Some practical languages scale back a bit from the full power
  - Let polymorphism, rank-2 polymorphism...

## Lists in system-F

- We can now type the List operations:

$$\text{nil} : \forall T. \text{List } T$$

$$\text{cons} : \forall T. T \rightarrow \text{List } T \rightarrow \text{List } T$$

$$\text{isnil} : \forall T. \text{List } T \rightarrow \text{bool}$$

$$\text{head} : \forall T. \text{List } T \rightarrow T$$

$$\text{tail} : \forall T. \text{List } T \rightarrow \text{List } T$$

- The list functions are just values in System-F, with polymorphic types
- instantiating is according to T-TypeApplication
- Pierce shows an encoding of the type `List T` as

$$\forall U. (T \rightarrow U \rightarrow U) \rightarrow U \rightarrow U$$

## Example

- Polymorphic `map` function

```
map =  $\Lambda T. \Lambda U.$ 
```

```
   $\lambda f: T \rightarrow U.$ 
```

```
    (fix ( $\lambda m: (\text{List } T) \rightarrow (\text{List } U).$ 
```

```
       $\lambda l: \text{List } T.$ 
```

```
        if isnil [T] l
```

```
        then nil [U]
```

```
        else cons [U] (f (head [T]) l))
```

```
                      (m (tail [U]) l))))
```

```
map :  $\forall T. \forall U. (T \rightarrow U) \rightarrow \text{List } T \rightarrow \text{List } U$ 
```

- Example use:

```
ls1 = cons [nat] 1 (cons [nat] 0 (cons [nat] 3 (nil [nat])))
```

```
ls2 = map [nat] [bool] ( $\lambda x:\text{nat}. \text{iszero } x$ ) ls1
```

# Properties of System F

- Progress and preservation still hold
- **Normalization** holds as well: evaluation of well-typed programs terminate

# What does “forall” really mean?

- In an universal type  $\forall T$ , we quantify over “all types”
- The definition of “all” is not necessarily clear
- **Predicative polymorphism**
  - $T$  ranges over simple types
- **Impredicative polymorphism**
  - $T$  ranges over simple types **and polymorphic types**
  - This is System F
- **type:type polymorphism**
  - $T$  ranges over all types, including itself!

# Predicative polymorphism

- Let  $T$  refer to the collection of all types. Then  $T$  is defined **after** all of its members have already been defined. That means that  $T$  does not contain any types that are defined in terms of  $T$ .
- Consider the polymorphic identity function:  
$$\text{id} = \Lambda T. \lambda x : T. x$$
- $\text{id} [\text{nat}]$ ,  $\text{id} [(\text{nat} \rightarrow \text{nat}) \rightarrow \text{bool}]$  etc. are shorthands for defining simply typed functions.
- Polymorphic types are “type schemes”, not real types
- Type inference decidable

## Example of difference between ML and System F

- In O'Caml we can write

```
let id a = a in (id 1, id "one")
```

- but not:

```
let makepair f = (f 1, f "one")
```

- Typing in System F:

```
let makepair : ( $\forall T. T \rightarrow T$ )  $\rightarrow$  (int, string)  
f :  $\forall T. T \rightarrow T = (f 1, f "one")$ 
```

# Impredicative polymorphism

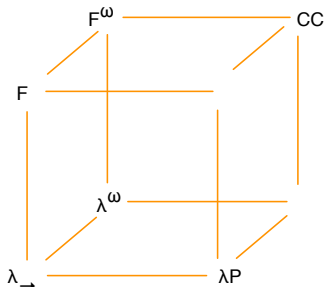
- Let  $T$  refer to the collection of all types.  $T$  contains types that are defined in terms of  $T$ .
- Applications with polymorphic types are valid:  
 $id = \Lambda T. \lambda x : T. x$   
 $id [\forall T. T \rightarrow T]$
- Self-application becomes possible
  - E.g. we applied the polymorphic identity function to its own type
- Type inferencing undecidable (Wells, 1994)
  - which is the reason for ML's let-polymorphism, rank-2 polymorphism, etc.

## type:type polymorphism

- Effectively allows type-functions
  - Arbitrary computations on types
- Type checking undecidable
- “... logical interpretation of the system meaningless, it is debatable whether such systems may nonetheless be useful in some situations as type systems for programming languages.”[ATTPL]

# Barendregt's lambda cube

- **Lambda cube** and **pure type systems** characterize differences in various systems
- Same type system “scheme” with variation in the abstraction rule
  - $\lambda_{\rightarrow}$  : term-term abstraction
  - $F$  : term-type abstraction
  - $\lambda P$  : type-term abstraction, dependent types
  - $\lambda^{\omega}$  : kinding, type operators
  - $F^{\omega}$  : higher order type-type abstraction
- Note, impredicative/predicative/type:type is a different direction of variability



# Outline

- 1 Introduction
- 2 Aside: general recursion
- 3 About System F
- 4 Existential types**
- 5 Abstract data types
- 6 Existential objects

# Existential types

- Existential types are a means for information hiding and abstraction
- Not a strict extension, remember from predicate logic:

$$\forall x.P(x) \equiv \neg(\exists x.\neg P(x))$$

- Indeed, existential types can be encoded as universal types
- Syntax:

$$\{\exists X, T\}$$

# Operational and logical views $\forall$

- Consider a term  $t$  with type  $\forall X. T$
- Logical view:
  - $t$  has value of type  $[S/X]T$  for any  $S$
- Operational view
  - $t$  is a **mapping** from type  $S$  to a term with the type  $[S/X]T$
  - This is how we defined the operational semantics of System F

Operational and logical views:  $\exists$ 

- Consider a term  $t$  with type  $\{\exists X, T\}$
- Logical view:
  - $t$  has value of type  $[S/X]T$  for some type  $S$
- Operational view:
  - $t$  is a pair of a type  $S$  and a term of type  $[S/X]T$
  - $S$  is **hidden**
  - We write the pair as  $\{*S, u\}$ , where  $*$  merely differentiates from term level tuples
- The notation  $\{\exists X. T\}$  (instead of  $\exists X. T$ ) emphasizes the latter view

## Intuition behind existentials

- Constructing a value of an existential type

$$\{*S, t\}$$

- A pair (**module**, **package**) consisting of a the hidden **representation type** (or **witness type**) and a term of that type
- The hidden type is not accessible from outside — type system enforces an abstraction barrier
- Consider the following package:

$$p = \{*\text{nat}, \{a = 1, b = \lambda x:\text{nat}. \text{pred } x\}\}$$

- What is its type?

## Intuition behind existentials

- Constructing a value of an existential type

$$\{*S, t\}$$

- A pair (**module**, **package**) consisting of a the hidden **representation type** (or **witness type**) and a term of that type
- The hidden type is not accessible from outside — type system enforces an abstraction barrier
- Consider the following package:

$$p = \{*\text{nat}, \{a = 1, b = \lambda x:\text{nat}. \text{pred } x\}\}$$

- What is its type?

$$\{ \exists X, \{a:X, b:X \rightarrow X\} \}$$

## Intuition behind existentials

- Constructing a value of an existential type

$$\{ *S, t \}$$

- A pair (**module**, **package**) consisting of a the hidden **representation type** (or **witness type**) and a term of that type
- The hidden type is not accessible from outside — type system enforces an abstraction barrier
- Consider the following package:

$$p = \{ *nat, \{ a = 1, b = \lambda x:nat. \text{pred } x \} \}$$

- What is its type?

$$\{ \exists X, \{ a:X, b:X \rightarrow X \} \}$$

- What about this type?

$$\{ \exists X, \{ a:X, b:X \rightarrow nat \} \}$$

## Intuition behind existentials

- Constructing a value of an existential type

$$\{ *S, t \}$$

- A pair (**module**, **package**) consisting of a the hidden **representation type** (or **witness type**) and a term of that type
- The hidden type is not accessible from outside — type system enforces an abstraction barrier
- Consider the following package:

$$p = \{ *nat, \{ a = 1, b = \lambda x:nat. \text{pred } x \} \}$$

- What is its type?

$$\{ \exists X, \{ a:X, b:X \rightarrow X \} \}$$

- What about this type?

$$\{ \exists X, \{ a:X, b:X \rightarrow nat \} \}$$

- No general solution, requires programmer annotation

# Annotating existential types

```
p = {*nat, {a = 1, b = λx:nat. pred x}}
    as {∃X, {a:X, b:X → X}}
```

- Now  $p$  has type:

```
{ ∃X, {a:X, b:X → X}}
```

- Another type can be given too:

```
p' = {*nat, {a = 1, b = λx:nat. pred x}}
     as {∃X, {a:X, b:X → nat}}
```

- And now  $p'$  has type:

```
{ ∃X, {a:X, b:X → nat}}
```

# Typing rules

$$\frac{\text{T-PackExistential} \quad \Gamma \vdash t : [U/X]T}{\Gamma \vdash \{ *U, t \} \text{ as } \{ \exists X, T \} : \{ \exists X, T \}}$$

- The big thing of existentials is that packages with different representation types can have the same existential type
  - This is what gives information hiding
- Example

```
p1 = { *nat, { a = 1, b = λx:nat. iszero x } }
      as { ∃X, { a:X, b:X → bool } }
```

```
p2 = { *bool, { a = false,
               b = λx:bool. if x then false else true } }
      as { ∃X, { a:X, b:X → bool } }
```

# Elimination rule

$$\begin{array}{c}
 \text{T-UnpackExistential} \\
 \frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2}
 \end{array}$$

- Unpacking an existential is comparable to **importing** or **opening** a module/package
- The existential becomes available but the representation type is not accessible
- Only the capabilities of the existential type are accessible
- Example

```
let {X,t} = p2 in (t.b t.a)
```

## Elimination rule

$$\frac{\text{T-UnpackExistential} \quad \Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2}$$

- Unpacking an existential is comparable to **importing** or **opening** a module/package
- The existential becomes available but the representation type is not accessible
- Only the capabilities of the existential type are accessible
- Example

```
let {X,t} = p2 in (t.b t.a)
```

```
$> true : bool
```

## Evaluation rules

E-Pack

$$\frac{t \rightarrow t'}{\{\ast T, t\} \text{ as } U \rightarrow \{\ast T, t'\} \text{ as } U}$$

E-Unpack

$$\frac{t_1 \rightarrow t'_1}{\text{let } \{X, x\} = t_1 \text{ in } t_2 \rightarrow \text{let } \{X, x\} = t'_1 \text{ in } t_2}$$

E-UnpackPack

$$\text{let } \{X, x\} = (\{\ast T, v\} \text{ as } U) \text{ in } t_2 \rightarrow [T/X][v/x]t_2$$

- The last rule is like module **linking**: symbolic names are replaced with the concrete components
- Note: an expression can get a more exact type with evaluation
- Evaluation rules don't care about typing, an ill-typed expression could be evaluated to a well-typed one

## Examples

- The type can only be used “abstractly”, representation type is not allowed to leak

```
t = {*nat, {a = 1, b = λx:nat. iszero x}
      as {∃ X, {a:X, b:X → bool}}}}
let {X, x} = t in pred x.a // error!
```

- Why is this a reasonable restriction?
- The type variable of the existential package can also be used in the scope of the unpacked package

```
let {X, x} = t in (λ y:X. x.b y) x.a
$> false : bool
```

- But it cannot be free in the resulting type:

```
let {X, x} = t in x.a // error
```

- Here, `x.a` would have type `X`, which is not in the typing context of the result

## More examples

- Consider the term:

$$t = \{*\text{nat}, \{a = 1, b = \lambda x:\text{nat}. \text{succ } x\}\}$$

- and these types given to  $t$ :

$$T_1 = \{\exists X, \{a:X, b:X \rightarrow \text{Nat}\}\}$$

$$T_2 = \{\exists X, \{a:X, b:X \rightarrow X\}\}$$

$$T_3 = \{\exists X, \{a:X, b:\text{Nat} \rightarrow X\}\}$$

$$T_4 = \{\exists X, \{a:\text{Nat}, b:\text{Nat} \rightarrow \text{Nat}\}\}$$

- Analyze. Remember, only allowed operations are those given by the type of the package

# Abstract data types with existentials

- Abstract data type:
  - 1 Name of the type **A**
  - 2 Representation type **T**
  - 3 Implementation of operations to create, manipulate and query values of the representation type
  - 4 **abstraction boundary**
- Outside the boundary, viewed as **A**, can only be manipulated with operations defined in the ADT
- Inside the boundary, viewed as **T**, all operations allowed

## Example

- A definition of an ADT

```

ADT counter =
  type Counter;
  representation nat;
  signature
    new : Counter;
    get  : Counter → nat;
    inc  : Counter → Counter;
  operations
    new = 1;
    get = λ i:nat.i;
    inc = λ i:nat.succ(i);
  
```

- Example use. The fact that `counter` is really a `nat` cannot be exploited

```
counter.get (counter.inc counter.new);
```

## Counter as an existential type

```

ADT counter =
  type Counter;
  representation nat;
  signature
    new : Counter;
    get  : Counter → nat;
    inc  : Counter → Counter;
  operations
    new = 1;
    get = λ i:nat.i;
    inc = λ i:nat.succ(i);

```

```

let {Counter, counter} = counterADT in
counter.get (counter.inc counter.new);
$> 2 : nat

```

```

counterADT =
  { *nat,
    {
      new = 1,
      get = λ i:nat. i,
      inc = λ i:nat. succ(i)
    }
  }
as
  {∃ Counter,
    {
      new: Counter,
      get: Counter → nat,
      inc: Counter → Counter
    }
  }

```

## Example continues

- After unpacking:

```
let {C, c} = <counter-package>
    in <rest-of-the-program>
```

both the term ( $C$ ) and type ( $c$ ) are fully usable in the rest of program

- Example:

```
let {C, c} = counterADT in
let add3 =  $\lambda x:C. c.inc (c.inc (c.inc x))$ 
    in c.get (add3 c.new)
```

```
%> 4 : nat
```

# Example continues

```

let {Cnt, cnt} = counterADT in
let {DoubleCounter, doublecounter} =
  { *Cnt,
    {
      new = cnt.new,
      get = λ i:Cnt. cnt,
      doubleinc = λ i:Cnt. i.inc (i.inc cnt)
    }
  }
as
{∃ DoubleCounter,
  {
    new: DoubleCounter,
    get: DoubleCounter → nat,
    doubleinc: DoubleCounter → DoubleCounter
  }
} in

doublecounter.get (doublecounter.doubleinc doublecounter.new);

%> 3 : nat

```

# Representation independence

- The key property that existential types give is **representation independence**
- The implementation of an ADT can change under the hoods without affecting type safety

```

counterADT =
  { *{x:nat},
    {
      new = {x=1},
      get = λ i:{x:nat}. i.x,
      inc = λ i:{x:nat}. {x=succ(i.x)}
    }
  }
as
{∃ Counter,
  {
    new: Counter,
    get: Counter → nat,
    inc: Counter → Counter
  }
}

```

# Existentials representing objects

- Existential types can abstract data into **objects**
- Different style as compared to ADTs
  - With ADT's, the package is opened up entirely, and the program outside of the package manipulates the representation type
  - With objects, the package is kept closed, operations are part of the package, and only those can get hold of the representation type

# Example

```
Counter = { $\exists X$ , {state:X, methods:{get:X $\rightarrow$  nat, inc:X $\rightarrow$  X}}};
```

```
c = {*nat,
     { state = 1,
       methods = { get =  $\lambda$  x:nat. x,
                   inc =  $\lambda$  x:nat. succ (x)}}
  as Counter;
```

```
let {X,body} = c in body.methods.get(body.state);
%> 1 : nat
```

```
sendget : Counter  $\rightarrow$  nat
sendget =  $\lambda$  c:Counter.
  let {X,body} = c in body.methods.get(body.state);
```

```
sendget c;
%> 1 : nat
```

- Note, **Counter** is not the representation type, but rather the entire existential type

# Increment

```
sendinc : Counter → Counter
sendinc = λ c:Counter.
  let {X, body} = c in
    { *X,
      {state = body.methods.inc(body.state),
        methods = body.methods}}
    as Counter;

add3 = λ c:Counter. sendinc (sendinc (sendinc c));
add3 : Counter → Counter
```

# Objects and ADTs