

Polymorphism

- Outline
 - *Extending the type system*
 - *Products*
 - *Disjoint unions*
 - *Existential types*



The polymorphic lambda calculus

e	$::=$	v	variables
		$\lambda x:t.e$	abstraction
		$e_1 e_2$	application
		$\Lambda X.e$	type abstraction
		$e[t]$	type application
t	$::=$	X	type variables
		$t_1 \rightarrow t_2$	function types
		$\forall X.t$	type quantification



Coding other primitives

$$\begin{aligned}\perp &\Leftrightarrow \forall X.X \\ \varphi \wedge \psi &\Leftrightarrow \forall X.(\varphi \Rightarrow \psi \Rightarrow X) \Rightarrow X \\ \varphi \vee \psi &\Leftrightarrow \forall X.(\varphi \Rightarrow X) \Rightarrow (\psi \Rightarrow X) \Rightarrow X \\ \exists x.\varphi &\Leftrightarrow \forall X.(\forall x.\varphi \Rightarrow X) \Rightarrow X \\ \exists X.\varphi &\Leftrightarrow \forall Y.(\forall X.\varphi \Rightarrow Y) \Rightarrow Y\end{aligned}$$



Proof interpretation

- A proof of $\varphi \wedge \psi$ is a proof of φ and a proof of ψ
- A proof of $\varphi \vee \psi$ is a proof of φ or it is a proof of ψ
- A proof of $\exists X.\phi$ is witness X and a proof of ϕ



Elimination (second-order) interpretation

- The program $\forall X.(\varphi \Rightarrow \psi \Rightarrow X) \Rightarrow X$ is a function that produces a proof of X given a function that produces a proof of X from proofs of φ and ψ .
- The program $\forall X.(\varphi \Rightarrow X) \Rightarrow (\psi \Rightarrow X) \Rightarrow X$ is a function that produces a proof of X given two functions, one to produce X from φ and another to produce X from ψ .
- The program $\forall Y.(\forall X.\varphi \Rightarrow Y) \Rightarrow Y$ produces a proof of Y given a function that produces a proof of Y from *any* proof of φ .



Propositions-as-types

Proposition	Type	Interpretation
$A \wedge B$	$A * B$	Products (tuples)
$A \vee B$	$A + B$	Disjoint unions (datatypes)
$\exists X.A$	$X : Type * A$ $\Sigma X: Type. A$	Dependent products



General properties

- Every type has a
 - *Constructor: the introduction form*
 - *Destructor: the elimination form*
- Usually, we'll only deal with binary type constructors
 - *Easy to generalize to arbitrary finite arities (for example tuples instead of pairs)*
- Side-effects won't change much



Simple products (pairs)

The *product* space $A * B$ is a space of pairs:

$$\frac{\Gamma \vdash a \in A \quad \Gamma \vdash b \in B}{\Gamma \vdash (a, b) \in (A * B)} \text{ pair intro}$$

The elimination forms are the projections $e.1$ and $e.2$

$$\frac{\Gamma \vdash x \in (A * B)}{\Gamma \vdash x.1 \in A} \text{ fst}$$

$$\frac{\Gamma \vdash x \in (A * B)}{\Gamma \vdash x.2 \in B} \text{ snd}$$



Product operational semantics

$$(a, b).1 \rightarrow a$$

$$(a, b).2 \rightarrow b$$



Products

```
let f (x, y) = x + y
val f : (int * int) → int
```

```
let f z =
  let x = fst z in
  let y = snd z in
    x + y
val f : (int * int) → int
```

```
let i = f (5, 17)
val i : int = 22
```

```
let f x y = x + y
val f : int → int → int
```

```
let g = f 5
val g : int → int
```

```
let i = g 17
val i : int = 22
```



“Currying”

Currying:

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

Uncurrying:

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



Currying in ML

Currying: $((A * B) \rightarrow C) \rightarrow (A \rightarrow B \rightarrow C)$

let f g = (**fun** x y \rightarrow g (x, y));;

val f : $((\alpha * \beta) \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma)$

Uncurrying: $(A \rightarrow B \rightarrow C) \rightarrow ((A * B) \rightarrow C)$

let f g = (**fun** (x, y) \rightarrow g x y);;

val f : $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow ((\alpha * \beta) \rightarrow \gamma)$



Notes on simple products

- $(A * B)$ describes the pairs (a, b)
- Larger arities are straightforward
 - $(A * B * C * D)$ contains (a, b, c, d) , etc.
- Records $\{ I1 : t1; I2 : t2; \dots; In = tn \}$
 - Just like tuples, but elements are named
 - Same for C
 - `struct {`
 - `int I1;`
 - `char *I2;`
 - `...`
 - `}`



(Tagged/Disjoint/Variant) unions

The *union* space $A + B$ is a disjoint space $A \cup B$, with elements $\mathbf{inl}(a)$ and $\mathbf{inr}(b)$.

$$\frac{\Gamma \vdash a \in A}{\Gamma \vdash \mathbf{inl}(a) \in (A + B)} \text{ inl}$$

$$\frac{\Gamma \vdash b \in B}{\Gamma \vdash \mathbf{inr}(b) \in (A + B)} \text{ inl}$$



Union elimination

The elimination forms does a case analysis:

$$\frac{\Gamma \vdash x \in (A + B) \quad \Gamma, y:A \vdash e_1 : t \quad \Gamma, z:B \vdash e_2 : t}{\Gamma \vdash (\mathbf{match} \ x \ \mathbf{with} \ \mathbf{inl}(y) \rightarrow e_1 \mid \mathbf{inr}(z) \rightarrow e_2) : t} \text{ match}$$

Operational semantics:

$$\begin{aligned} &(\mathbf{match} \ \mathbf{inl}(a) \ \mathbf{with} \ \mathbf{inl}(y) \rightarrow e_1 \mid \mathbf{inr}(z) \rightarrow e_2) \rightarrow e_1[a/x] \\ &(\mathbf{match} \ \mathbf{inr}(b) \ \mathbf{with} \ \mathbf{inl}(y) \rightarrow e_1 \mid \mathbf{inr}(z) \rightarrow e_2) \rightarrow e_2[b/x] \end{aligned}$$



ML example

```
type number =  
  Int of int  
  | Float of float
```

```
let number_plus_int x i =  
  match x with  
    Int x → Int (x + i)  
  | Float x → Float (x +. (float_of_int i))
```



“match” is the second-order operator

type $(\alpha, \beta) t = A \text{ of } \alpha \mid B \text{ of } \beta$

let $f \ x \ g \ h =$

match x **with**

$A \ y \rightarrow g \ y$

$\mid B \ z \rightarrow h \ z;;$

val $f : (\alpha, \beta) t \rightarrow (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow \gamma$



Product spaces

let $f \times g =$
 match x with
 $(y, z) \rightarrow g \ y \ z;;$

val $f : (\alpha * \beta) \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \gamma$



Notes on union

- The normal implementation is a “tagged” value
 - *type $t = A \text{ of } \text{int} \mid B \text{ of } \text{float} \mid C \text{ of } \text{string}$*
 - $A \ 1 == (0, 1)$
 - $B \ 3.14 == (1, 3.14)$
 - $C \ \text{“hello”} == (2, \text{“hello”})$
- Generalization to arbitrary arity is straightforward
- C unions are problematic
 - *They have no tag*
 - *So they are not constructive—but perhaps still sensible in some classical (noncomputable) theory*



Existential types

- Proof interpretation: a proof of $\exists X.\varphi$ is a *witness* X , and a proof of φ
- This suggests that the programs that have existential type are *pairs* $(t, e) \in \exists X.A$, where t is a type, and $e \in A[t/X]$



Identity example

- $\exists X.X$
 - $(\mathbb{Z}, 0)$
 - $(A \rightarrow A, \lambda x:A.x)$



An abstract value

- $\exists X. X \rightarrow \mathbb{Z}$
 - $(\mathbb{Z}, \lambda i. i)$
 - $(A * \mathbb{Z}, \lambda x. (x.2))$
 - $(A * (A \rightarrow \mathbb{Z}), \lambda x. (x.2)(x.1))$



forall-exists/Exists-forall types

- $\forall X. \exists Y. (X \rightarrow Y)$
 - $\Lambda X. (X, \lambda x: X. x)$
 - $\Lambda X. (\mathbb{Z}. \lambda x: X. 1)$
- $\exists Y. \forall X. (X \rightarrow Y)$
 - $(\mathbb{Z}, \Lambda X. \lambda x: X. 1)$



ML abstract data types

```
module type Set =  
sig  
  type t  
  val empty : t  
  val add : t → int → t  
  val mem : t → int → bool  
end
```



List implementation

```
module type Set =  
sig  
  type t  
  val empty : t  
  val add : t → int → t  
  val mem : t → int → bool  
end
```

```
module IntSet : Set =  
struct  
  type t = int list  
  let empty = []  
  let add s i = i :: s  
  let mem s i = List.mem i s  
end
```



Bitmask implementation

```
module type Set =  
sig  
  type t  
  val empty : t  
  val add : t → int → t  
  val mem : t → int → bool  
end
```

```
module IntSet : Set =  
struct  
  type t = int  
  let empty = 0  
  let add s i = s | (1 « i)  
  let mem s i = (s & (1 « i)) != 0  
end
```



Existential interpretation

```
module type Set =  
sig  
  type t  
  val empty : t  
  val add : t → int → t  
  val mem : t → int → bool  
end
```

```
∃t. {  
  empty : t  
  * add : t → ℤ → t  
  * mem : t → ℤ → ℬ  
}
```



List implementation

(\mathbb{Z} list,
let empty = []
let add s i = i :: s
let mem s i = List.mem i s)

$\exists t.
 empty : *t*
 * *add* : *t* → \mathbb{Z} → *t*
 * *mem* : *t* → \mathbb{Z} → \mathbb{B}
}$



“Abstract” interpretations

- Usually, we intend this to be an abstract data type
 - We can't figure out what the set representation is
 - We are not allowed to write code that depends on the set representation
- Two interpretations
 - “Strong” sums (products): the type is visible
 - “Weak” sums: the type is invisible (not computable)



Strong/weak interpretations

- **Strong** sums $\exists X.t$
 - The elements are pairs (t', e) where $e \in t[t' / X]$
 - Often written $\Sigma X.t$, or $x: \text{Type} * t$
- **Weak** sums $\exists X.t$
 - The elements are programs e where $e \in t[t' / X]$ for *some* type t'
 - This is really a (non-disjoint) union $\bigcup_X t$



Syntax for existential

$e ::= \dots$
| $\{t_1, e\} \text{ as } t_2$ pack
| $\text{let}\{X, x\} = e_1 \text{ in } e_2$ unpack

$t ::= X$ type variables
| $t_1 \rightarrow t_2$ function types
| $\forall X.t$ polymorphism
| $\exists X.t$ abstraction



Operational semantics

let{ X, x } = ($\{t_1, v_1\}$ **as** t_2) **in** $e_2 \rightarrow e_2[t_1/X, v_1/x]$

Note, the v_1 expression must be a value.



Typing rules

$$\frac{\Gamma \vdash e : t[U/X]}{\Gamma \vdash (\{U, e\} \text{ as } \exists X.t) : \exists X.t} \text{ pack}$$

$$\frac{\Gamma \vdash e_1 : \exists X.t_1 \quad \Gamma, X, x:t_1 \vdash e_2 : t_2}{\Gamma \vdash (\text{let}\{X, x\} = e_1 \text{ in } e_2) : t_2} \text{ unpack}$$

