Lesson 5
Simple extensions of the typed lambda calculus

1/20/03

Chapter 11

A toolkit of useful constructs

- Base types
- Unit (nullary product)
- Sequencing and wild cards
- Type ascription
- Let bindings
- Pairs, tuples and records
- Sums, variants, and datatypes
- General recursion (fix, letrec)

Base Types

Base types are primitive or atomic types.

E.g. Bool, Nat, String, Float, ...

Normally they have associated ***intro*** and ***elimination*** syntactic forms, with associated rules, or alternatively sets of predefined constants and functions for creating and manipulating elements of the type. These rules or sets of constants provide an *interpretation* of the type.

Uninterpreted types can be used in expressions like $\lambda x{:}\ A.\ x$
but no values of these types can be created.

Type Unit

Type:
  Unit

Terms:
  unit

Rules:
  $\Gamma \vdash \text{unit} : \text{Unit}$

The type Unit has just one value: unit. It is typically used as the return type of a function that is used for effect (e.g. assignment to a variable).

In ML, unit is written as “( )”.

It plays a role similar to void in C, Java.

Derived forms

Sequencing: $t_1; t_2$

Can be treated as a basic term form, with its own evaluation and typing rules (call this λ^E, the external language):

$$\frac{t_1 \rightarrow t_1'}{t_1; t_2 \rightarrow t_1'; t_2} \quad \text{(E-Seq)} \qquad \text{unit}; t_2 \rightarrow t_2 \quad \text{(E-SeqNext)}$$

$$\frac{\Gamma \vdash t_1 : \text{Unit} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2} \quad \text{(T-Seq)}$$

Sequencing as Derived Form

Sequencing Can be also be treated as an abbreviation:

t1; t2 $=_{def}$ (λx: Unit. t2) t1 (with x fresh)

This definition can be used to map λ^E to the internal language λ^I consisting of λ with Unit.

Elaboration function:

e: $\lambda^E \rightarrow \lambda^I$

e(t1; t2) = (λx: Unit. t2) t1

e(t) = t otherwise

Elaboration theorem

Theorem: For each term t of λ^E we have

$t \rightarrow^E t'$ *iff* $e(t) \rightarrow^I e(t')$

$\Gamma \vdash^E t : T$ *iff* $\Gamma \vdash^I e(t) : T$

Proof: induction on the structure of t.

Type ascription

Terms:

t as T

Eval Rules:

$$v \text{ as } T \rightarrow v \qquad \text{(E-Ascribe)}$$

$$\frac{t \rightarrow t'}{t \text{ as } T \rightarrow t' \text{ as } T} \qquad \text{(E-Ascribe1)}$$

Type Rules:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ as } T : T} \qquad \text{(T-Ascribe)}$$

Let expressions

Terms:

let x = t1 in t2

Eval Rules: $\text{let } x = v \text{ in } t \rightarrow [x \mapsto v]\, t$ (E-LetV)

$$\frac{t_1 \rightarrow t_1'}{\text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t_1' \text{ in } t_2} \quad \text{(E-Let)}$$

Type Rules:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}\, T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(T-App)}$$

Let expressions

Let as a derived form

$e(\text{let } x = t_1 \text{ in } t_2) = (\lambda x : T.\ t_2)\ t_1$

but where does T come from?

Could add type to let-binding:

$\text{let } x : T = t_1 \text{ in } t_2$

or could use type checking to discover it.

Pairs

Types:
$T_1 \times T_2$

Terms:
$\{t, t\} \mid t.1 \mid t.2$

Values:
$\{v, v\}$

Eval Rules: (i = 1,2)

$\{v_1, v_2\}.i \rightarrow v_i$ (E-PairBeta i)

$$\frac{t_1 \rightarrow t_1'}{t_1.i \rightarrow t_1'.i}$$ (E-Proj i)

$$\frac{t_1 \rightarrow t_1'}{\{t_1, t_2\} \rightarrow \{t_1', t_2\}}$$ (E-Pair1)

$$\frac{t_2 \rightarrow t_2'}{\{v_1, t_2\} \rightarrow \{v_1, t_2'\}}$$ (E-Pair2)

Pairs - Typing

Typing Rules:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2} \qquad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.i : T_i} \qquad \text{(T-Proj i)}$$

Naive semantics: Cartesian product

$$T_1 \times T_2 = \{(x,y) \mid x \in T_1 \text{ and } y \in T_2\}$$

Properties of Pairs

1. access is positional -- order matters

   (3, true) ≠ (true, 3) $Nat \times Bool \neq Bool \times Nat$

2. evaluation is left to right

   (print “x”, raise Fail)   prints and then fails
   (raise Fail, print “x”)   fails and does not print

3. projection is “strict” -- pair must be fully evaluated

Tuples

Type constructors:

$\{T_1, T_2, ..., T_n\}$ or $T_1 \times T_2 \times ... \times T_n$

Tuple terms

$\{t_1, t_2, ..., t_n\}$ or $(t_1, t_2, ..., t_n)$

Projections

$t : \{T_1, T_2, ..., T_n\}$ => $t.i : T_i$ $(i = 1, ..., n)$

Properties of Tuples

- Evaluation and typing rules are the natural generalizations of those for pairs.
- Evaluation is left-to-right.
- Tuples are fully evaluated before projection is evaluated.
- Pairs are a special case of tuples.

Examples:

{true, 1, 3} : {Bool, Nat, Nat} (or Bool × Nat × Nat)

{true, 1} : {Bool, Nat} (equivalent to Bool × Nat)

Records

- Records are “labelled tuples”.

  {name = “John”, age = 23, student = true}
  : {name: String, age: Nat, student: Bool}

- Selection/projection is by label, not by position.

  let x = {name = “John”, age = 23, student = true}
  in if x.student then print x.name else unit

  t : {name: String, age: Nat, student: Bool}
  => t.name : String, t.age : Nat, t.student : Bool

- Components of a record are called fields.

Records - Evaluation

- Evaluation of record terms is left to right, as for tuples.
- Tuples are fully evaluated before projection is evaluated.
- Order of fields matters for evaluation

let x = ref 0
 in {a = !x, b = (x := 1; 2)}
→* {a = 0, b = 2}

let x = ref 0
 in {b = (x := 1; 2), a = !x}
→* {b = 2, a = 1}

Records - Field order

- Different record types can have the same labels:

  {name: String, age: Nat} $\neq$ {age: Nat, name: Bool}

- What about order of fields? Are these types equal?

  {name: String, age: Nat} = {age: Nat, name: String} ?

  We can choose either convention. In SML, field order is not relevant, and these two types are equal. In other languages, and in the text (for now), field order is important. and these two types are different.

Extending Let with Patterns

let {name = n, age = a} = find(key)
 in if a > 21 then name else "anonymous"

The left hand side of a binding in a let expression can be a record pattern, that is matched with the value of the rhs of the binding.

We can also have tuple patterns:
  let (x,y) = coords(point) in ... x ... y ...

See Exercise 11.8.2 and Figure 11-8.

Sum types

Types:

T1 + T2

Terms:

inl t

inr t

case t of inl x => t | inr x => t

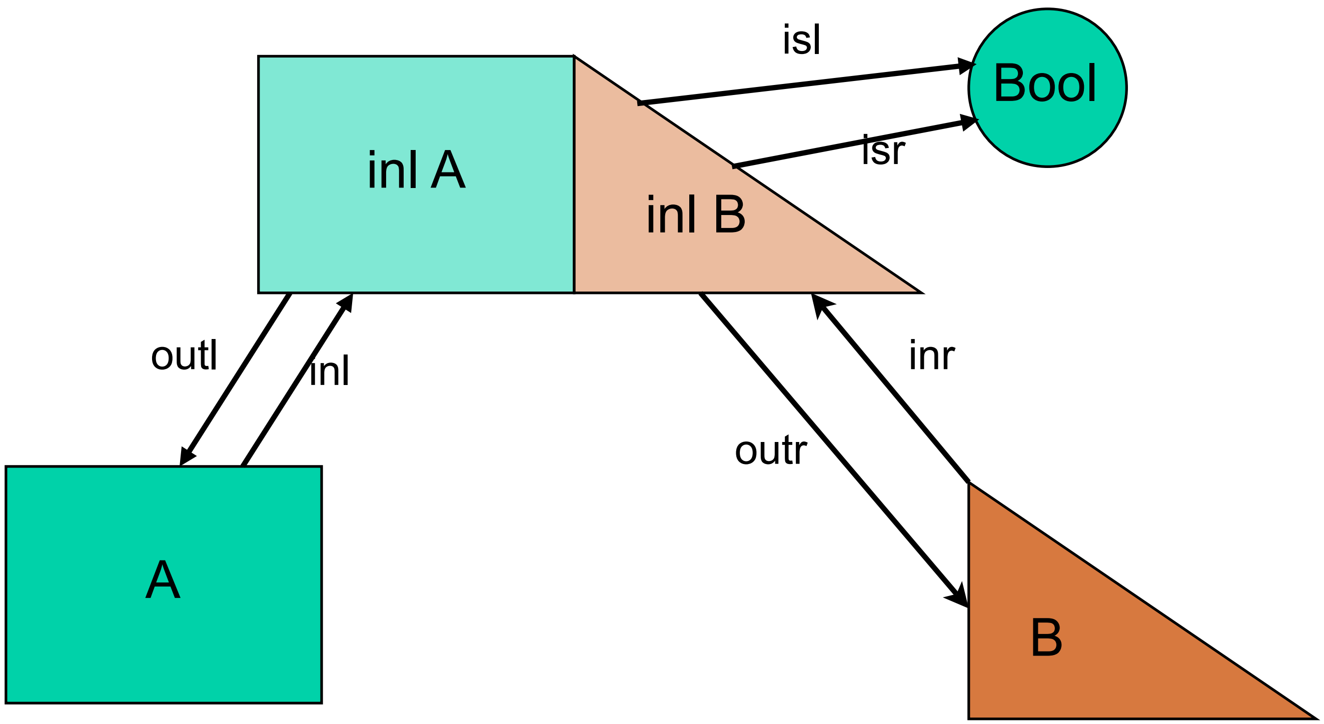
Values:

inl v

inr v

Sum types

Sum types represent disjoint, or discriminated unions



Sum types

$A + B = \{(1,a) \mid a \in A\} \cup \{(2,b) \mid b \in B\}$

inl a = (1,a)
inr b = (2,b)
outl (1,a) = a
outr(2,b) = b
isl (1,a) = true; isl (2,b) = false
isr (1,a) = false; isr (2,b) = true

case z of inl x => t1 | inr y => t2 =
  if isl z then (λ x. t1)(outl z) else (λ y. t2)(outr z)

Sums - Typing

$$\frac{\Gamma \vdash t1 : T1}{\Gamma \vdash \text{inl } t1 : T1 + T2} \quad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash t2 : T2}{\Gamma \vdash \text{inr } t2 : T1 + T2} \quad \text{(T-Inr)}$$

$$\frac{\Gamma \vdash t : T1 + T2 \qquad \Gamma, x1 : T1 \vdash t1 : T \qquad \Gamma, x2 : T2 \vdash t2 : T}{\Gamma \vdash \text{case } t \text{ of inl } x1 \Rightarrow t1 \mid \text{inr } x2 \Rightarrow t2 : T} \quad \text{(T-Case)}$$

Typing Sums

Note that terms do not have unique types:

inl 5 : Nat + Nat   and   inl 5 : Nat + Bool

Can fix this by requiring type ascriptions with inl, inr:

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 : T_1 + T_2} \quad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \text{inr } t_2 \text{ as } T_1 + T_2 : T_1 + T_2} \quad \text{(T-Inr)}$$

Labeled variants

Could generalize binary sum to n-ary sums, as we did going from pairs to tuples. Instead, go directly to labeled sums:

type NatString = <nat: Nat, string: String>

a = <nat = 5> as NatString
b = <string = “abc”> as NatString

```
λx: NatString . case x
                  of <nat = x> => numberOfDigits x
                   | <string = y> => stringLength y
```

Option

An option type is a useful special case of labeled variants.

type NatOption = <some: Nat, none: Unit>

someNat = λx: Nat . <some = x> as NatOption
: Nat → NatOption

noneNat = <none = unit> as NatOption : NatOption

half = λx: Nat . if equal(x, 2 * (x div 2)) then someNat(x div 2)
else noneNat
: Nat → NatOption

Enumerations

Enumerations are another common form of labeled variants.
They are a labeled sum of several copies of Unit.

type WeekDay = <monday: Unit, tuesday: Unit, wednesday: Unit,
thursday: Unit, friday: Unit>

monday = <monday = unit> as WeekDay

type Bool = <true: Unit, false: Unit>
true = <true = unit> as Bool
false = <false = unit> as Bool

ML Datatypes

ML datatypes are a restricted form of labeled variant type
+ recursion + parameterization

datatype NatString = **Nat** of Nat | **String** of String
fun size x = case x
of **Nat** n => numberOfDigits n
| **String** s => stringLength s

datatype NatOption = **Some** of Nat | **None**
datatype ‘a Option = **Some** of ‘a | **None** (‘a is a type variable)
datatype Bool = **True** | **False**
datatype ‘a List = **Nil** | **Cons** of ‘a * ‘a List (recursive datatype)

General Recursion

The fixed point combinator (p. 65), can't be defined in $\lambda\rightarrow$.
So we need to defined a special fix operator.

Terms: fix t

Evaluation

$$\text{fix } (\lambda x{:}\ T.\ t) \rightarrow [x \mapsto (\text{fix}(\lambda x{:}\ T.\ t))]\ t \qquad \text{(E-FixBeta)}$$

$$\frac{t_1 \rightarrow t_1'}{\text{fix } t_1 \rightarrow \text{fix } t_1'} \qquad \text{(E-Fix)}$$

General Recursion - Typing

Typing

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \text{fix } t : T} \quad \text{(T-Fix)}$$

The argument t of fix is called a generator.

Derived form:

letrec x: T = t_1 in t_2
  $=_{def}$ let x = fix(λx: T. t_1) in t_2

Mutual recursion

The generator is a term of type T→T for some T, which is typically a function type, but may be a tuple or record of function types to define a family of mutually recursive functions.

```
ff = λieio: {iseven: Nat → Bool, isodd: Nat → Bool} .
        {iseven = λx: Nat . if iszero x then true
                            else ieio.isodd (pred x),
         isodd  = λx: Nat . if iszero x then false
                            else ieio.iseven (pred x)}
  : T → T   where T is {iseven: Nat → Bool, isodd: Nat → Bool}

r = fix ff : {iseven: Nat → Bool, isodd: Nat → Bool}
iseven = r.iseven  :  Nat → Bool
```

Lists

Type:

List T

Terms t:

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

Values v:

nil [T]
cons [T] v v

Lists - Evaluation

Eval rules:

isnil[S] (nil[T]) → true (E-IsnilNil)

head[S] (cons[T] v_1 v_2) → v_1 (E-HeadCons)

tail[S] (cons[T] v_1 v_2) → v_2 (E-TailCons)

plus usual congruence rules for evaluating arguments.

Lists - Typing

$$\Gamma \vdash \text{nil}[T] : \text{List } T \qquad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : \text{List } T}{\Gamma \vdash \text{cons}[T]\ t_1\ t_2 : \text{List } T} \qquad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash t : \text{List } T}{\Gamma \vdash \text{isnil}[T]\ t : \text{Bool}} \qquad \text{(T-Isnil)}$$

$$\frac{\Gamma \vdash t : \text{List } T}{\Gamma \vdash \text{head}[T]\ t : T} \qquad \text{(T-Head)}$$

$$\frac{\Gamma \vdash t : \text{List } T}{\Gamma \vdash \text{head}[T]\ t : \text{List } T} \qquad \text{(T-Tail)}$$