Lesson 5
Simple extensions of the typed lambda calculus

1/24-2/02

Chapter 11

A toolkit of useful constructs

- Base types
- Unit
- 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 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:
Γ |- unit : 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: t1; t2

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

$$\frac{t1 \rightarrow t1'}{t1;\ t2 \rightarrow t1';\ t2} \quad \text{(E-Seq)} \qquad unit;\ t2 \rightarrow t2 \quad \text{(E-SeqNext)}$$

$$\frac{\Gamma \mid\!- \ t1 : Unit \qquad \Gamma \mid\!- \ t2 : T2}{\Gamma \mid\!- \ t1;\ t2 : T2} \quad \text{(T-Seq)}$$

1/31/02 Lesson 5 5

Sequencing as Derived Form

Sequencing Can be also be treated as an abbreviation:

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

This definition can be used to map λ^E to the internal language λ^I consisting of $\lambda_{\rightarrow}$ with Unit.

Elaboration function

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

$e(t1;\ t2) = (\lambda x{:}\ Unit.\ t2)\ t1$

$e(t) = t$ otherwise

1/31/02 Lesson 5 6

Elaboration theorem

Theorem: For each term t of □^{E} we have

t □ ^{E} t' iff e(t) □ ^{I} e(t')

□ |-^{E} t : T iff □ |-^{I} e(t) : T

Proof: induction on the structure of t.

Type ascription

Terms:

t as T

Eval Rules:

v as T □ v (E-Ascribe)

$$\frac{\text{t1 □ t1'}}{\text{t1 as T □ t1' as T}}$$ (E-Ascribe1)

Type Rules:

$$\frac{\text{□ |- t1 : T}}{\text{□ |- t1 as T : T}}$$ (T-Ascribe)

Let expressions

Terms:
let x = t1 in t2

Eval Rules: let x = v in t $\rightarrow$ [x ⇨ v]t (E-LetV)

$$\frac{t1 \rightarrow t1'}{\text{let } x = t1 \text{ in } t2 \rightarrow \text{let } x = t1' \text{ in } t2} \quad \text{(E-Let)}$$

Type Rules:

$$\frac{\Gamma \mid\!- t1 : T1 \qquad \Gamma, x{:}\ T1 \mid\!- t2 : T2}{\Gamma \mid\!- \text{let } x = t1 \text{ in } t2 : T2} \quad \text{(T-App)}$$

1/31/02 Lesson 5 9

Let expressions

Let as a derived form

e(let x = t1 in t2) = (λx : T. t2) t1

but where does T come from?

Could add type to let-binding:

let x: T = t1 in t2

or could use type checking to discover it.

1/31/02 Lesson 5 10

Pairs

Types: $T1 \times T2$

Terms: {t, t} | t.1 | t.2

Values: {v, v}

Eval Rules: (i = 1,2)

$\{v1,v2\}.i \rightarrow v1$ (E-PairBeta i)

$$\frac{t1 \rightarrow t1'}{t1.i \rightarrow t1'.i} \quad \text{(E-Proj i)}$$

$$\frac{t1 \rightarrow t1'}{\{t1, t2\} \rightarrow \{t1', t2\}} \quad \text{(E-Pair1)}$$

$$\frac{t2 \rightarrow t2'}{\{v1, t2\} \rightarrow \{v1, t2'\}} \quad \text{(E-Pair2)}$$

Pairs - Typing

Typing Rules:

$$\frac{\Gamma \mid\!- \; t1 : T1 \qquad \Gamma \mid\!- \; t2 : T2}{\Gamma \mid\!- \; \{t1, t2\} : T1 \times T2} \quad \text{(T-Pair)}$$

$$\frac{\Gamma \mid\!- \; t : T1 \times T2}{\Gamma \mid\!- \; t.i : Ti} \quad \text{(T-Proj i)}$$

Naive semantics: Cartesian product

$T1 \times T2 = \{(x,y) \mid x \in T1 \text{ and } y \in T2\}$

Properties of Pairs

1. access is positional -- order matters

    (3, true) ≠ (true, 3)   Nat × Bool ≠ Bool × 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

1/31/02 Lesson 5 13

Tuples

Type constructors:

{T1, T2, ..., Tn}   or   T1 × T2 × ... × Tn

Tuple terms

{t1, t2, ..., tn}   or   (t1, t2, ..., tn)

Projections

t : {T1, T2, ..., Tn}   =>   t.i : Ti (i = 1, ..., n)

1/31/02 Lesson 5 14

Properties of Tuples

- Evaluation and typing rules are the natural generalizations of those for tuples.
- 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}
```

1/31/02 Lesson 5 17

Records - Field order

- Different record types can have the same labels:
  {name: String, age: Nat} ≠ {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.

1/31/02 Lesson 5 18

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

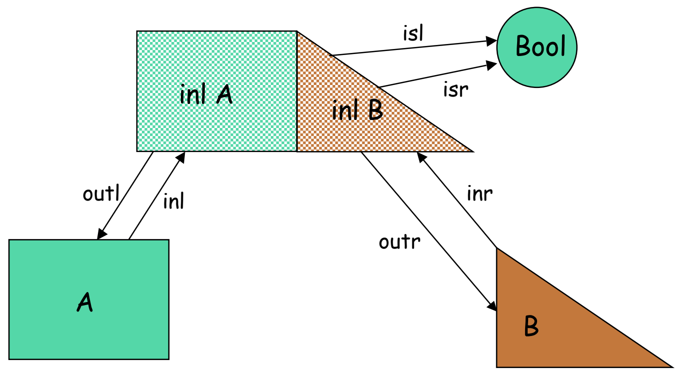
Types: T1 + T2

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



1/31/02 Lesson 5 21

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 (λx. t1)(outl z) else (λy. t2)(outr z)

1/31/02 Lesson 5 22

Sums - Typing

$$\frac{\square \; |\text{-}\; t1 : T1}{\square \; |\text{-}\; \text{inl}\ t1 : T1 + T2} \quad \text{(T-Inl)}$$

$$\frac{\square \; |\text{-}\; t2 : T2}{\square \; |\text{-}\; \text{inr}\ t2 : T1 + T2} \quad \text{(T-Inr)}$$

$$\frac{\square \; |\text{-}\; t : T1 + T2 \qquad \square, x1: T1 \; |\text{-}\; t1 : T \qquad \square, x2: T2 \; |\text{-}\; t2 : T}{\square \; |\text{-}\; \text{case}\ t\ \text{of inl}\ x1 \Rightarrow t1 \mid \text{inr}\ x2 \Rightarrow t2 : T}$$

(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:

$$\frac{\square \; |\text{-}\; t1 : T1}{\square \; |\text{-}\; \text{inl}\ t1\ \text{as}\ T1 + T2 : T1 + T2} \quad \text{(T-Inl)}$$

$$\frac{\square \; |\text{-}\; t2 : T2}{\square \; |\text{-}\; \text{inr}\ t2\ \text{as}\ T1 + T2 : T1 + T2} \quad \text{(T-Inr)}$$

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 anonther 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 type defn)

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

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

$$\frac{t1 \rightarrow t1'}{\text{fix } t1 \rightarrow \text{fix } t1'} \qquad \text{(E-Fix)}$$

General Recursion - Typing

Typing

$$\frac{\Gamma \vdash t1 : T1 \rightarrow T1}{\Gamma \vdash \text{fix } t1 : T1} \qquad \text{(T-Fix)}$$

The argument t1 of fix is called a generator.

Derived form:

letrec x: T1 = t1 in t2
$=_{def}$ let x = fix(λx: T1. t1) in t2

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
```

1/31/02 Lesson 5 31

Lists

Type: List T

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

Values: nil[T]
cons[T] v v

1/31/02 Lesson 5 32

Lists - Evaluation

Eval rules:

isnil[S](nil[T]) → true (E-IsnilNil)

head[S](cons[T] v1 v2) → v1 (E-HeadCons)

tail[S](cons[T] v1 v2) → v2 (E-TailCons)

plus usual congruence rules for evaluating arguments.

Lists - Typing

$$\Gamma \mid\!- \text{nil[T1]} : \text{List T1} \qquad \text{(T-Nil)}$$

$$\frac{\Gamma \mid\!- \text{t1} : \text{T1} \qquad \Gamma \mid\!- \text{t2} : \text{List T1}}{\Gamma \mid\!- \text{cons[T1] t1 t2} : \text{List T1}} \qquad \text{(T-Cons)}$$

$$\frac{\Gamma \mid\!- \text{t} : \text{List T1}}{\Gamma \mid\!- \text{isnil[T1] t} : \text{Bool}} \qquad \text{(T-Isnil)}$$

$$\frac{\Gamma \mid\!- \text{t} : \text{List T1}}{\Gamma \mid\!- \text{head[T1] t} : \text{T1}} \quad \text{(T-Head)} \qquad \frac{\Gamma \mid\!- \text{t} : \text{List T1}}{\Gamma \mid\!- \text{head[T1] t} : \text{List T1}} \quad \text{(T-Tail)}$$