Administrative Announcements

- No lecture on Tuesday, March 12, but Midterm 1 will be given out that day (to be returned within 24 hours).
- Thursday, March 14 will be a guest lecture on effect systems by Manuel Fahndrich.

History of Types

- Fortran: floats, ints, arrays
- COBOL: records (fields were named without hierarchy)
- LISP: type predicates – atomp, listp (dynanmically typed)
- Algol-68: types as a set of constrctors, selectors, predicates
- C: Types only for defining constructors and selectors. No abstract arrays. Little (dynamic or static) type checking.
- Pascal: like C, a reaction to the complexity of Algol 68
- Scheme: dynamically typed, like LISP.
- Modula and Euclid: abstract data types
- CLU: abstract polymorphic types, exceptions, strong typing
- Smalltalk: inheritance
- ML: strong typed with type inference, exceptions

Types

What is a type?

Types add redundancy to a programming language:

- documentation
- eliminates some incorrect programs (early error detection)
- better information for compile-time optimization
- necessarily innacurate – both overly conservative and and incomplete

Dimensions of Types

Dynamic vs. Static

Explicit vs. Implicit

Expressive vs. Restrictive

Expression are similar to those in FL, but have types added to all introductions of identifiers.

$$E \in \text{Exp}$$
$$T \in \text{Type}$$

Functions, for example, are typed and statically checked.

$$\begin{array}{rcl}
E & ::= & (\texttt{lambda}\ ((I_{formal}\ T)^*) E_{body}) \\
 & | & (E_{rator}\ E_{rand}) \\
 & | & (\texttt{let}\ ((I_{name} T\ E_{def}) *)\ E_{body}) \\
 & | & (\texttt{letrec}\ ((I_{name} T\ E_{def})^*)\ E_{body}) \\
 & | & (\texttt{the}\ T\ E) \\
 & | & (\texttt{tlet}\ ((I\ T)^*)\ E)
\end{array}$$

$$\begin{array}{rcl}
T & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{sym} \mid \texttt{string} \mid I \\
 & | & (\texttt{->}\ (T^*)\ T_{body})
\end{array}$$

Function Types

```
(the (-> (int) int)
     (lambda ((x int)) (* x x)))

(the (-> (bool int int) bool)
     (lambda ((p bool) (a int) (b int))
          (if p (= a b) #t)))

(the int (letrec ((fact (-> (int) int)
                        (lambda ((n int))
                            (if (= n 0)
                                1
                                (* n (fact (- n 1)))))))
             (fact 5)))
```

Records and Tagged Unions

$$
\begin{array}{rcl}
E & ::= & (\texttt{record}\ ((I\ T\ E)^*)) \\
 & | & (\texttt{select}\ I\ E) \\
 & | & (\texttt{one}\ T\ I_{tag}\ E_{val}) \\
 & | & (\texttt{tagcase}\ E_{disc}\ (I_{tag}\ I_{val}\ E_{val})+) \\
 & | & (\texttt{tagcase}\ E_{disc}\ (I_{tag}\ I_{val}\ E_{val})+\ (\texttt{else}\ E_{else}))
\end{array}
$$

$$
\begin{array}{rcl}
T & ::= & (\texttt{recordof}\ (I\ T)^*) \\
 & | & (\texttt{oneof}\ (I\ T^*))
\end{array}
$$

```
(the (oneof (finite int) (infinite bool) (nan unit))
     (one (oneof finite int) (infinite bool) (nan unit))
              finite -1))
```

Record and Tagged Union Example

```
(tlet ((extint (oneof (finite int) (infinite bool)
                      (nan unit))))
  (the (recordof
          (mk-finite (-> (int) extint))
          (ext+ (-> (extint extint) extint)))
    (record (mk-finite
              (-> (int) extint)
              (lambda ((n int)) (one extint finite n)))
            (ext+
              (-> (extint extint) extint)
              (lambda ((x extint) (y extint))
                (tagcase x
                   (finite xint
                      (tagcase y (finite yint
                                   (mk-finite (+ x y)))
                                 (infinite ysign
                                   (mk-infinite ...)
     ...)))))))))
```

Recursive Types

$$
\begin{array}{rcl}
T & ::= & (\texttt{recof}\ \ I\ T) \\
  & | & (\texttt{rectype}\ \ (I\ T)\ T_{body})
\end{array}
$$

Simple recursion:

```
(recof int-tree
  (onof
     (leaf int)
     (node (recordof (left int-tree) (right int-tree)))))
```

Recursive Types

Mutual recursion – a red-black tree:

```
(rectype ((red-level
              (recordof (label int)
                        (left black-level)
                        (right black-level)))
           (black-level
             (oneof
               (leaf null)
               (node (recordof (label int)
                               (left either-level)
                               (right either-level)))))
          (either-level
             (oneof
               (red-either red-level)
               (black-either black-level)))))
      black-level)
```

Type Equivalence

Scheme/X allows for new type identifiers:

$$
\begin{aligned}
E & ::= \ldots \\
& \mid \; (\texttt{tlet}\ ((I\ T)^*)\ E)
\end{aligned}
$$

This is typical of most strongly typed languages. However, notions of type equality differ between languages.

1. Name equivalence (strongest)

   T1 is equivalent only to T1, i.e., a type of the same name

2. Instance Equivalence

   ```
   type T1 = int;
   type T2 = T1;                 T1 and T2 are equivalent.
   type T3 = T1;                 T1 and T3 are equivalent.
                                 T2 and T3 are not.
   ```

3. Structural equivalence (weakest)

   T1 and T2 are equivalent if they expand (after possibly infinite expansion) to the same structure.

Type Equivalence in Scheme/X

$$T = T \qquad [reflexive]$$

$$\frac{T_1 = T_2}{T_2 = T_1} \qquad [symmetric]$$

$$\frac{T_1 = T_2 \ \& \ T_2 = T_3}{T_1 = T_3} \qquad [transitive]$$

$$\frac{\forall\ i\ (S_i = T_i)\ \&\ S_{body} = T_{body}}{(\texttt{->}\ \ (S_1 \ldots S_n) S_{body}) = (\texttt{->}\ \ (T_1 \ldots T_n) T_{body})} \qquad [application]$$

$$\frac{\begin{array}{c}\{J_1, \ldots, J_n\} = \{I_1, \ldots, I_n\} \\ \forall\ i, j\ (J_j = I_i)\ \Rightarrow S_j = T_i\end{array}}{\begin{array}{c}(\texttt{recordof}\ \ (J_1\ S_1) \ldots (J_n\ S_n)) = \\ \quad (\texttt{recordof}\ \ (I_1\ T_1) \ldots (I_n\ T_n))\end{array}} \qquad [records]$$

Tagged unions are similar to records. The language also has cells, vectors and lists with the obvious (equality) rules.

Type Equivalence of Recursive Types

$$(\texttt{recof}\ \ I\ T) = (\texttt{recof}\ \ I_{new}\ [I_{new}/I]T)$$
$$\text{where } I_{new} \text{ isnot free in } T \qquad [recof - alpha]$$

$$(\texttt{recof}\ \ I\ T) = [(\texttt{recof}\ \ I\ T)/I]T$$
$$[recof - beta]$$

But this does not give "infinite unrolling" equivalence. Two possible fixes:

1. Prove by induction, i.e., assume two types are equivalent until disproven.
2. Compute a normal form for each type expression.

Recursive Type Example

```
(recof intlist1
  (oneof (cons-cell (recordof (car int) (cdr intlist1)))
         (nil unit)))

(recof intlist2
  (oneof (cons-cell (recordof (car int) (cdr intlist2)))
         (nil unit)))

(recof (cons-cell
  (recordof (car int)
            (cdr (recof int-list3
                   (oneof (cons-cell
                            (recordof (car int)
                                      (cdr intlist3)))
                          (nil #u)))))))
```

Typing Rules

The type rules look like inference rules and axioms for SOS, but they are assertions about types. Something that has a provable type is considered *well-typed.*

$$\vdash\ N :\ int \qquad [int]$$

$$\vdash\ B :\ bool \qquad [bool]$$

$$A[I :\ T] \vdash I :\ T \qquad [var]$$

$$\frac{A \vdash E :\ T}{A \vdash (\texttt{the}\ \ T\ \ E) :\ T}\ [the]$$

Scheme/X Type rules

$$\frac{A \vdash \; E_1 : \; bool \; ; \; A \vdash E_2 : \; T \; ; \; A \vdash E_3 : \; T \; ;}{A \vdash \; (\texttt{if} \;\; E_1 \; E_2 \; E_3) : \; T} \quad [if]$$

$$\frac{A[I_1 : T_1, \; \ldots I_n : T_n] \vdash E_B : \; T_B}{\begin{array}{c} A \vdash \; (\texttt{lambda} \;\; ((I_1 \; T_1) \ldots (I_n \; T_n))E_B) \;\; : \\ (\texttt{->} \;\; (T_1 \ldots T_n) \; T_B) \end{array}} \quad [lambda]$$

$$\frac{\begin{array}{c} A \vdash E_P : \; (\texttt{->} \;\; (T_1 \ldots T_n) \; T_B) \\ \forall i (A \vdash E_i : \; T_i) \end{array}}{A \vdash \; (E_P \; E_1 \; \ldots E_n) \; : \; T_B} \quad [call]$$