ML and Typechecking - Datatypes

- Can define new data types and define functions over these data types using pattern-matching.

Feature exists in all modern functional languages (Hope, Miranga, Haskell, etc).

Eases programming as compared with Lisp and Scheme.

Types in Scheme and ML

- Scheme - all data types (lists, records, etc) were basically defined over a single data type, lists. Scheme has dynamic typing.

- ML - static typing, so we need to introduce separate data types. Saw lists and tuples (records). What about tree data types with different node types in a tree? (In Scheme these would still be lists, possibly with the car of the list is a symbol tagging the node type and the remainder of the list included any subtrees of that node.)

Side Issue - Records in ML

A record is a tuple with named fields. For example, `(real, real)` can be represented as `{re:real, im:real}`. We can refer to fields of the record using the field name:

```
# <field label> <record value>
```

Examples:

```
{name="Jane Doe", age = 28, married =true };
  val it = {age=28,married=true,name="Jane Doe"}
    age:int, married:bool, name:string}
# name it
  val it = "Jane Doe" : string
```

More familiar syntax, i.e., `it.name` is reserved for other purposes (field in a module).

Type Definition in ML

1) A type name is an abbreviation for a type expression:

```
type complex = {re:real, im:real};
  type complex = {re:real, im:real}
fun add (x:complex) (y:complex) :  complex
= {re = (# re x) + (# re y), im = (# im x)
+ (# im y)} ;
  val add = fn : complex -> complex -> complex
add { re=4.5, im=1.0 } { re=2.3, im=8.9 } ;
  val it = {re=6.8, im=9.9} : complex
```

Note:

- There is no data abstraction - the `add` procedure accesses both of its arguments as records, even though they are declared as `complex`
- Arguments to `add` do not have to be declared `complex`, as long as they have the same structure as `complex`. (ML uses `structural equivalence`.)

Type Definition in ML (Cont'd)

2) Can define enumerated types, which creates new type constructors and new data constructors:

```
datatype days = sun | mon | tue | wed
      | thu | fri | sat ;
```

type days

con sun : days

...

con sat : days Type constructor is `days` and data constructors (new constants that have been dynamically added to the program by the declaration) are `sun`, `mon`, ... `sat`.

Can use pattern matching to operate on these datatypes:

Type Definition - Example

```
fun what_to_do sun = "rest"
  | what_to_do mon = "program"
  | what_to_do tue = "program"
  | what_to_do wed = "program"
  | what_to_do thu = "program"
  | what_to_do fri = "party"
  | what_to_do sat = "shop"
  val what_to_do = fn : days -> string
```

This version has a lot of tiresome repetition. Can get avoid it by supplying a default case:

```
fun what_to_do sun = "rest"
  | what_to_do fri = "party"
  | what_to_do sat = "shop"
  val what_to_do = fn : days -> string
```

Type Definition in ML - Patterns

Datatype declarations can declare more than just enumerated types, and patterns can be very elaborate. We can define new types in terms of existing types:

```
datatype int_pair = pair of int * int
  type int_pair
    con pair : int * int -> int_pair

fun proj1 (pair p) = (#1 p);
  val proj1 = fn : int_pair -> int
```

Notes:

- Unlike the `type` declaration, this introduces a new type constructor `int_pair` which only matches with itself. This is *name equivalence*.
- Data constructors can be parameterized by values, and applications can also be patterns.

Notes on Patterns, Cont'd

- However, procedure being applied in a pattern application must be a data constructor name.

- Patterns can contain variables, but these are *defining occurences*. In definition of `proj1`, the variable `p` is declared in the pattern and used on the right hand side. - Variables can occur at most once in a pattern.

Patterns are used to define functions that can extract parts of any *observable* value. A pattern is composed of data constructors (possibly applied to patterns), variables and the special underscore character.

Patterns, Cont'd

Operations for forming lists, records, tuples - data constructors. Patterns can be used to extract components of these data structures.

```
val (x, y) = (17, true);
  val x = 17 : int
  val y = true : bool   fun proj1 (x, _) = x;
  val proj1 = fn: 'a * 'b -> 'a   fun car ( a ::
_) = a
| car nil = ...?
  car = fn : 'a list -> 'a   fun length ( _::  1)
= 1 + (length 1)
| length nil = 0;
  val length = fn :  'a list -> int
```

Can also have "deep" patterns:

```
fun cadr (_ ::  ( a ::  _) ) = a
| cadr [ _ ] = ...?
| cadr nil = ...?
  val cadr = fn : 'a list -> 'a
```

Polymorphic Data Structures

Type variables allow us to define polymorphic data structures:

```
datatype ('a, 'b) pair = pair of 'a * 'b
  type pair
  con pair : 'a * 'b -> ('a, 'b) pair

fun proj1 ( pair (x, _) ) = x;
  val proj1 = fn : ('a, 'b) pair -> 'a
```

Note that here we use the pairing notation to signify a type constructor of two type arguments

Recursive Datatypes

This is the true power of datatype declarations.

```
  datatype 'a mylist = mynil | mycons of 'a *
('a mylist)
    type mylist
    cons mynili : 'a mylist
    con mycons : 'a * ('a mylist ) -> 'a mylist
  length (mycons ( _, 1)) = 1 + (length 1)
  | length mynil = 0;
    val length = fn : 'a mylist -> int
```

Any tree structure can be defined as a datatype, with a data constructor in the definition for each different node type in the tree. Any datatype which does not contain arrow types is types is also an *eqtype*, i.e., based on the structure of a datatype definition, the compiler can automatically build an equiality predicate for that datatype.

Datatypes - Evaluation

Most important observation - datatypes can be recursive. This allows us to have data structures of unbounded size (otherwise lists would be the only data structures with this property).

Note that untyped λ-calculus terms are really terms of a special type in ML:

```
datatype U = lambda of U -> U ;
  type U
  con lambda : (U -> U) -> U

fun apply (lambda f) x = f x ;
  val apply = fn : U -> (U -> U)
```

Example (Sethi, p. 360)

Datatype P *bitree*

```
datatype bitree = leaf
| nonleaf of bitree * bitree
```

Here we have two value constructors, *leaf* and *nonleaf*.

Value constructors can appear within exprssions as constants if they are parameterless and as functions if they take parameters.

What do the following expressions correspond to?

`leaf`

`nonleaf (leaf, leaf)]]` `nonleaf (leaf, nonleaf (leaf, leaf))`

`nonleaf (nonleaf (nonleaf (leaf, leaf), leaf), leaf)`

Example (Cont'd)

Let's define functions on binary trees.

Count leaves:

```
  fun leafcount(leaf) = 1
  | leafcount(nonleaf(s, t)) = leafcount(s) +
leafcount(t);
```

Is the tree a leaf?

```
  fun isleaf(leaf) = true
  | isleaf(nonleaf(_)) = false;
```

Extract right subtree

```
  fun right(nonleaf(s, t)) = t;
```

What happens if tree is `leaf`?

ML - Review

Covered chapter 8 and 9.1-9.5 of Sethi

- ML is a strongly typed language which does static typechecking.

- An expression is not correct if ML cannot figure out its type.

- Differentiates between records (heterogeneous) and lists (homogeneous).

- Allows parametric polymorphism - ability to define functions to operate on data of any type and still be able to typecheck them.

ML - Review (cont'd)

- Uses structural equavalence.
- Has type variables.
- Overloading, or ad-hoc polymorphism, conflicts with parametric.
- Allows definition of new types, as enumerated constants or as recursive types.
- Has pattern matching in functions.
- Uses pattern matching to process user-defined types.