Inductive Data Types

Lars-Henrik Eriksson

Functional Programming 1

Original slides by Tjark Weber

Today

- New names for old types
- New types
  - Enumeration types
  - Tagged unions
  - Polymorphic data types
  - Inductive data types
    - Trees

New Names for Old Types

Type Bindings

The keyword **type** gives a new name to an existing type.

The simplest form is

type identifier = type_expression

For instance,

```
> type float = real;
type float

> type mylist = (char * int) list;
type mylist = (char * int) list
```

Type Bindings (cont.)

The new name becomes *synonymous* with the type expression. Both can be used interchangeably.

For instance,

```
> [(#"A", 1)] : mylist;
val it = [(#"A", 1)]: mylist

> ([(#"A", 1)] : mylist) =
    ([(#"A", 1)] : (char * int) list);
val it = true: bool
```

Parametric Type Bindings

Types can be *polymorphic*, i.e., contain type variables. Therefore, type bindings can take type variables as parameters.

The general form of a type binding is

type (type_variables) identifier = type_expression

For instance,

```
> type ('a, 'b) assoc_list = ('a * 'b) list;
type ('a, 'b) assoc_list = ('a * 'b) list
```

Parametric Type Bindings (cont.)

All type variables that appear in the type expression (i.e., on the right) must be mentioned as a parameter (i.e., on the left).

```
> type predicate = ’a -> bool;
Error-’a has not been declared in type declaration
```

If there is just one parameter, the parentheses are optional.

```
> type ’a predicate = ’a -> bool;
type ’a predicate = ’a -> bool
```

Type variables may be instantiated with types (as usual).

```
> op< : (int * int) predicate;
val it = fn: (int * int) predicate
```

New Types

Enumeration Types

The keyword **datatype** declares a *new* type. Here, we declare an *enumeration type*, i.e., a type that has a finite number of values C1, . . . , Cn:

datatype identifier $=$ C1 | ... | Cn

For instance,

```
> datatype direction = North | South | East | West;
datatype direction = East | North | South | West

> North;
val it = North: direction
```

Enumeration Types: Example

The types `bool` and `unit` each have a finite number of values. They could (in principle) be declared as enumeration types as follows:

```
datatype bool = true | false;

datatype unit = ();
```

Enumeration Types: Exercise

If we compare two integers a and b, then either (i) $a < b$, (ii) $a = b$, or (iii) $a > b$.

1. Declare an enumeration type `order` that has three values: `LESS`, `EQUAL`, `GREATER`.
2. Write a function `compare: int * int -> order` such that `compare (a, b)` returns `LESS` if `a < b`, `EQUAL` if `a = b`, and `GREATER` if `a > b`.

Enumeration Types: Exercise (cont.)

Solution:

```
datatype order = LESS | EQUAL | GREATER

(* compare (a, b)
   TYPE: int * int -> order PRE: true POST: ... EX: ...
 *)
fun compare (a, b) =
  if a < b then
    LESS
  else if a = b then
    EQUAL
  else (* a > b *)
    GREATER
```

`order` and `Int.compare` are already declared in the SML Basis Library.

Constructors

In a datatype declaration

$$\textbf{datatype}\ \text{identifier}\ = \text{C1} \mid \ldots \mid \text{Cn}$$

C1, . . . , Cn are called **constructors**.

- Coding convention: constructors begin with an uppercase letter (while functions and values begin with a lowercase letter).

Constructor Patterns

Constructors can be used in patterns. A constructor matches only itself.

For instance,

```
(* opposite d
   TYPE: direction -> direction
   PRE: true
   POST: the direction opposite d
   EXAMPLES: opposite North = South
 *)
fun opposite North = South
  | opposite South = North
  | opposite East = West
  | opposite West = East
```

Constructor Patterns (cont.)

Constructors can be used in patterns. A constructor matches only itself.

For instance,

```
(* sign n
   TYPE: int -> int
   PRE: true
   POST: ~1 if n<0, 0 if n=0, 1 if n>0
   EXAMPLES: sign 42 = 1
 *)
fun sign n =
  case Int.compare (n, 0) of
    LESS => ~1
  | EQUAL => 0
  | GREATER => 1
```

Beyond Enumeration Types

Constructors with Arguments

Constructors in a datatype declaration may take arguments. These are indicated with they keyword **of** followed by the type of the argument.

datatype identifier = C1 **of** type1 | ... | Cn **of** typeN

For instance,

```
> datatype rational = Rat of int * int;
datatype rational = Rat of int * int

> Rat (2,3);
val it = Rat (2, 3): rational
```

Constructors with Arguments: Pattern Matching

Constructors that take an argument can be used in patterns (together with a pattern for the argument).

For instance,

```
(* qadd (x, y)
   TYPE: rational * rational -> rational
   PRE: x and y are rational numbers
        (with denominator <> 0)
   POST: the sum of x and y
   EXAMPLES: qadd (Rat (1,2), Rat (1,3)) = Rat (5,6)
 *)
fun qadd (Rat (a,b), Rat (c,d)) =
  Rat (a*d + b*c, b*d)
```

Example: Geometric Shapes

A type that models circles (of a given radius), squares (of a given side length) and triangles (of three given side lengths):

```
datatype shape = Circle of real
               | Square of real
               | Triangle of real * real * real
```

Note that Triangle(a,b,c) does not represent a real triangle in case the largest of the numbers a, b and c is greater than the sum of the other two numbers.

Constructing values of type shape:

```
> Circle 1.0;
val it = Circle 1.0: shape
> Square (1.0 + 1.0);
val it = Square 2.0: shape
> Triangle (3.0, 4.0, 5.0);
val it = Triangle (3.0, 4.0, 5.0): shape
```

Example: Geometric Shapes (cont.)

A function that computes the area of a geometric shape:

```
(* area s
   TYPE: shape -> real
   PRE: If s is Triangle(a,b,c) then it must represent
        actual triangle.
   POST: the area of s
   EXAMPLES: area (Circle 1.0) = 3.141592654
 *)
fun area (Circle r) = Math.pi * r * r
  | area (Square l) = l * l
  | area (Triangle (a, b, c)) =
      let val s = (a + b + c) / 2.0
      in (* Heron's formula *)
        Math.sqrt (s * (s-a) * (s-b) * (s-c))
      end
```

The Type of Constructors

A constructor that takes no arguments has the declared type.

```
> North;
val it = North: direction
```

A constructor that takes an argument has a function type.

```
> Circle;
val it = fn: real -> shape
> Square;
val it = fn: real -> shape
> Triangle;
val it = fn: real * real * real -> shape
```

However, constructors (unlike functions) can be used in patterns!

Tagged Unions/Sum Types

Types declared via

datatype identifier = C1 **of** type1 | ... | Cn **of** typeN

are also known as **tagged unions** or **sum types**.

We can think of values of this type as either being a value of type 1 or ... or a value of type *N*, that is *tagged* with a constructor that indicates which type the value has.

Polymorphic Data Types

The argument types of constructors can be *polymorphic*, i.e., contain type variables. Therefore, datatype declarations can take type variables as parameters.

The general form of a datatype declaration is

$$\textbf{datatype}\ (\text{ type\_variables })\ \ \text{identifier}\ \ \begin{array}{l} = \text{C1}\ \textbf{of}\ \text{type1} \\ \mid\ \ \ldots \\ \mid\ \text{Cn}\ \textbf{of}\ \text{typeN} \end{array}$$

Example: 'a option

For instance,

```
> datatype ('a) option = NONE | SOME of 'a;
datatype 'a option = NONE | SOME of 'a

> NONE;
val it = NONE: 'a option
> SOME;
val it = fn: 'a -> 'a option
> SOME 42;
val it = SOME 42: int option
> SOME "foo";
val it = SOME "foo": string option
> SOME [];
val it = SOME []: 'a list option
```

The Option Type

We can think of values of type `'a option` as representing either a single value or zero values of type `'a`.

Type `'a option` is useful to model partial functions (i.e., functions that normally return a value of type `'a`, but may not do so for some argument values).

For instance, not every string corresponds to an integer value:

```
> Int.fromString "42";
val it = SOME 42: int option
```

```
> Int.fromString "foo";
val it = NONE: int option
```

`'a option` is more explicit than exceptions. (It requires callers to deal with the `NONE` case explicitly.) This may or may not be desirable.

`'a option` and related functions are declared in the SML Basis Library.

Polymorphic Data Types (cont.)

All type variables that appear in an argument type (i.e., on the right) must be mentioned as a parameter (i.e., on the left).

```
> datatype option = NONE | SOME of 'a;
Error-'a has not been declared in type declaration
```

If there is just one parameter, the parentheses are optional.

```
> datatype 'a option = NONE | SOME of 'a;
datatype 'a option = NONE | SOME of 'a
```

Type variables may be instantiated with types (as usual).

```
> NONE : int option;
val it = NONE: int option
```

Inductive Data Types

Inductive Data Types

The argument types of constructors may refer to (instances of) the data type that is being declared.

That is, in a datatype declaration

```
datatype ( type_variables ) identifier  = C1 of type1
                                        | ...
                                        | Cn of typeN
```

the expressions `type1`, ...typeN may contain `identifier` applied to (the correct number of) type parameters.

Example: `'a list`

The type `'a list` of lists (with elements from `'a`) is an inductive type.

1. Base case:
   `[] : 'a list`
2. Inductive step:
   If `x : 'a` and `xs : 'a list`, then `x::xs : 'a list`.

`'a list` could (in principle) be declared as follows:

```
datatype 'a list = [] | :: of 'a * 'a list
```

Example: Arithmetic Expressions

For instance, $1 + 2$, $3 \cdot 4 + 5$, $\ldots$

Let us define arithmetic expressions (that involve addition and multiplication over integers) inductively:

1. Base case:
   - Each integer is an arithmetic expression (aexp).
2. Inductive step:
   - If e_1 and e_2 are aexps, then $e_1 + e_2$ is an aexp.
   - If e_1 and e_2 are aexps, then $e_1 \cdot e_2$ is an aexp.

Example: Arithmetic Expressions

For instance, $1 + 2$, $3 \cdot 4 + 5$, ...

Let us define arithmetic expressions (that involve addition and multiplication over integers) inductively:

1. Base case:
   - Each integer is an arithmetic expression (aexp).
2. Inductive step:
   - If e_1 and e_2 are aexps, then $e_1 + e_2$ is an aexp.
   - If e_1 and e_2 are aexps, then $e_1 \cdot e_2$ is an aexp.

```
datatype aexp = Int of int
              | Plus of aexp * aexp
              | Times of aexp * aexp
```

Example: Evaluation of Arithmetic Expressions

```
datatype aexp = Int of int
              | Plus of aexp * aexp
              | Times of aexp * aexp

(* eval e
   TYPE: aexp -> int
   PRE: true
   POST: the value of e
   EXAMPLES: eval (Plus (Int 1, Int 2)) = 3
 *)
```

Example: Evaluation of Arithmetic Expressions

```
datatype aexp = Int of int
              | Plus of aexp * aexp
              | Times of aexp * aexp

(* eval e
   TYPE: aexp -> int
   PRE: true
   POST: the value of e
   EXAMPLES: eval (Plus (Int 1, Int 2)) = 3
 *)
(* VARIANT: size of e *)
fun eval (Int i) = i
  | eval (Plus (x, y)) = eval x + eval y
  | eval (Times (x, y)) = eval x * eval y
```

Data Types: Structural Equality

Equality of datatype values is *structural*. Two values of the same datatype are equal if (and only if) they are built by the same constructor applied to equal arguments.

For instance,

```
  > Int 1 = Int 1;
```

Data Types: Structural Equality

Equality of datatype values is *structural*. Two values of the same datatype are equal if (and only if) they are built by the same constructor applied to equal arguments.

For instance,

```
> Int 1 = Int 1;
val it = true: bool
```

```
> Int 1 = Int 2;
```

Data Types: Structural Equality

Equality of datatype values is *structural*. Two values of the same datatype are equal if (and only if) they are built by the same constructor applied to equal arguments.

For instance,

```
> Int 1 = Int 1;
val it = true: bool
```

```
> Int 1 = Int 2;
val it = false: bool
```

```
> Plus (Int 1, Int 2) = Int 3;
```

Data Types: Structural Equality

Equality of datatype values is *structural*. Two values of the same datatype are equal if (and only if) they are built by the same constructor applied to equal arguments.

For instance,

```
> Int 1 = Int 1;
val it = true: bool

> Int 1 = Int 2;
val it = false: bool

> Plus (Int 1, Int 2) = Int 3;
val it = false: bool

> Plus (Int 1, Int 2) = Plus (Int 1, Int 2);
```

Data Types: Structural Equality

Equality of datatype values is *structural*. Two values of the same datatype are equal if (and only if) they are built by the same constructor applied to equal arguments.

For instance,

```
> Int 1 = Int 1;
val it = true: bool

> Int 1 = Int 2;
val it = false: bool

> Plus (Int 1, Int 2) = Int 3;
val it = false: bool

> Plus (Int 1, Int 2) = Plus (Int 1, Int 2);
val it = true: bool
```

Not All Data Types Are Equality Types

Suppose type `t` is declared via

$$\begin{array}{rl} \textbf{datatype}\ (\ \text{type\_variables}\ )\ \mathtt{t} = & \text{C1}\ \textbf{of}\ \text{type1} \\ | & \ldots \\ | & \text{Cn}\ \textbf{of}\ \text{typeN} \end{array}$$

An instance `(t1, ..., tK) t` is an equality type if `t1`, `...`, `tK` are equality types *and* the instances of `type1`, `...`, `typeN` are equality types.

For instance,

```
> Circle 1.0 = Square 1.0;
```

Not All Data Types Are Equality Types

Suppose type `t` is declared via

$$
\begin{aligned}
\textbf{datatype}\ (\ \text{type\_variables}\ )\ \mathtt{t}\ &= \text{C1}\ \textbf{of}\ \text{type1} \\
&\mid\ \ldots \\
&\mid\ \text{Cn}\ \textbf{of}\ \text{typeN}
\end{aligned}
$$

An instance `(t1, ..., tK) t` is an equality type if `t1`, `...`, `tK` are equality types *and* the instances of `type1`, `...`, `typeN` are equality types.

For instance,

```
> Circle 1.0 = Square 1.0;
Error-Type error in function application.
   Function: = : ''a * ''a -> bool
   Argument: (Circle 1.0, Square 1.0) : shape * shape
   Reason: Can't unify ''a to shape (Requires equality type)
```

Not All Data Types Are Equality Types (cont.)

```
datatype 'a option = NONE | SOME of 'a

> NONE = SOME 1.0;
```

Not All Data Types Are Equality Types (cont.)

```
datatype 'a option = NONE | SOME of 'a

> NONE = SOME 1.0;
Error-Type error in function application.
   Function: = : ''a option * ''a option -> bool
   Argument: (NONE, SOME 1.0) : ''a option * real option
   Reason: Can't unify ''a to real (Requires equality type)

> NONE = SOME 1;
```

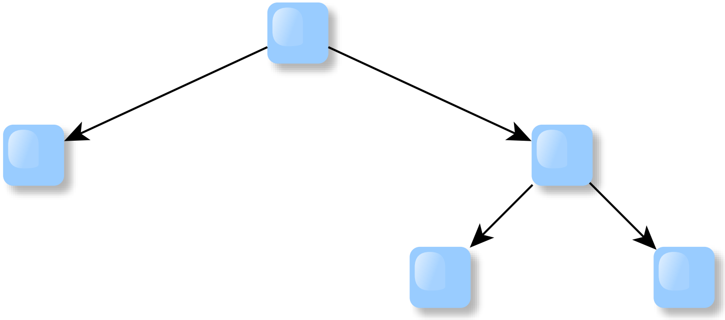
Not All Data Types Are Equality Types (cont.)

```
datatype 'a option = NONE | SOME of 'a

> NONE = SOME 1.0;
Error-Type error in function application.
   Function: = : ''a option * ''a option -> bool
   Argument: (NONE, SOME 1.0) : ''a option * real option
   Reason: Can't unify ''a to real (Requires equality type)

> NONE = SOME 1;
val it = false: bool
```

Trees

Full Binary Trees

A tree where each inner node has exactly two children is called a **full binary tree**.



Full Binary Trees: Inductive Definition

Full binary trees (with labels of type `'a`) can be defined inductively:

1. Base case:
   Each value of type `'a` is a full binary tree, namely a leaf.
2. Inductive step:
   If x is of type `'a` and t_1 and t_2 are full binary trees, then Node (t_1, x, t_2) is a full binary tree.

In SML:

Full Binary Trees: Inductive Definition

Full binary trees (with labels of type `'a`) can be defined inductively:

1. Base case:

   Each value of type `'a` is a full binary tree, namely a leaf.

2. Inductive step:

   If x is of type `'a` and t_1 and t_2 are full binary trees, then Node (t_1, x, t_2) is a full binary tree.

In SML:

```
datatype 'a fbtree = Leaf of 'a
                   | Node of 'a fbtree * 'a * 'a fbtree
```

Full Binary Trees: Examples

```
datatype 'a fbtree = Leaf of 'a
                   | Node of 'a fbtree * 'a * 'a fbtree
```

Some full binary trees in SML:

```
> Leaf "Grandfather";
val it = Leaf "Grandfather": string fbtree

> Node (Leaf "Grandfather", "Father", Leaf "Grandmother");
val it = Node (Leaf "Grandfather", "Father",
   Leaf "Grandmother"): string fbtree
```

Pattern Matching: Example

Functions over trees—like functions over inductive data types in general—are usually defined using pattern matching and recursion.

For instance,

```
(* root_value t
   TYPE: 'a fbtree -> 'a
   PRE: true
   POST: the value at t's root node
   EXAMPLES: root_value (Leaf "foo") = "foo"
 *)
```

Pattern Matching: Example

Functions over trees—like functions over inductive data types in general—are usually defined using pattern matching and recursion.

For instance,

```
(* root_value t
   TYPE: 'a fbtree -> 'a
   PRE: true
   POST: the value at t's root node
   EXAMPLES: root_value (Leaf "foo") = "foo"
 *)
fun root_value (Leaf x) = x
  | root_value (Node (_, x, _)) = x
```

Tree Height in SML

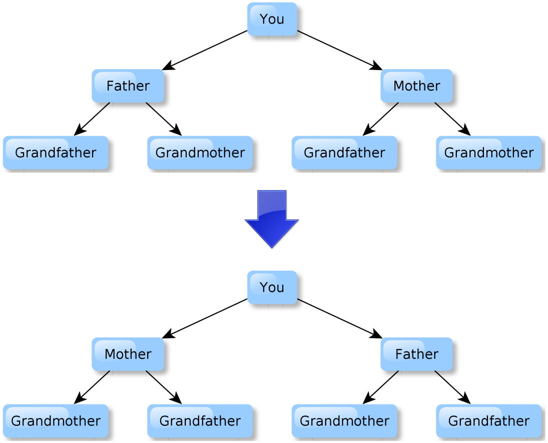
```
(* height t
   TYPE: 'a fbtree -> int
   PRE: true
   POST: the height of t
   EXAMPLES: height (Leaf "foo") = 0
 *)
```

Tree Height in SML

```
(* height t
   TYPE: ’a fbtree -> int
   PRE: true
   POST: the height of t
   EXAMPLES: height (Leaf "foo") = 0
 *)
(* VARIANT: size of t *)
fun height (Leaf _) = 0
  | height (Node (l, _, r)) =
      1 + Int.max (height l, height r)
```

Mirror Image

Let’s write a function that “mirrors” a full binary tree by (recursively) exchanging left and right subtrees. For instance,



Mirror Image in SML

```
(* mirror t
   TYPE: 'a fbtree -> 'a fbtree
   PRE: true
   POST: the mirror image of t
   EXAMPLES: mirror (Node (Leaf 1, 2, Leaf 3)) =
                Node (Leaf 3, 2, Leaf 1)
 *)
```

Mirror Image in SML

```
(* mirror t
   TYPE: 'a fbtree -> 'a fbtree
   PRE: true
   POST: the mirror image of t
   EXAMPLES: mirror (Node (Leaf 1, 2, Leaf 3)) =
                Node (Leaf 3, 2, Leaf 1)
 *)
(* VARIANT: size of t *)
fun mirror (Leaf x) = Leaf x
  | mirror (Node (l, x, r)) = Node (mirror r, x, mirror l)
```