

## Algebraic types with Field Labels

As an alternative to the *positional* notation shown above, algebraic types' fields can be assigned *labels* (as in C and Pascal).

For example,

```
> data Time
> = HMS { hr, min :: Int, sec :: Float }
```

The field labels serve as selector functions; their types are

```
hr, min :: Time -> Int
sec     :: Time -> Float
```

so toSeconds could be defined thus:

```
> toSeconds :: Time -> Float
> toSeconds t
> = fromInt(60 * (60 * hr t + min t)) + sec t
```

(the first definition of toSeconds would also work).

Compare with C:

```
struct Time { int hr; int min; float sec; };
float toSeconds( Time t ) {
    return (60 * (60 * t.hr + t.min)) + t.sec;
}
```

Field labels are better than positional notation for types that have numerous fields —especially when software modifications require changes in the type's definition.

Algebraic types defined using field labels can have multiple constructors. An example:

```
> data Shape = Point { loc :: (Int,Int) }
>             | Square { loc :: (Int,Int),
>                       size :: Int }
>             | Ellipse { loc :: (Int,Int),
>                         size :: Int,
>                         eccentricity :: Float
>                       }
> pt :: Shape
> pt = Point { loc = (3,4) }
> sq :: Shape
> sq = Square { loc = (7,8), size = 25 }
> e1 :: Shape
> e1 = Ellipse { size = 10, eccentricity = 0.2 }
```

Applying field-label functions:

```
Main> loc pt
(3,4)
Main> size sq
25
Main> size pt
Program error: {Shape_Square_size pt}
Main> loc e1
Program error: Ellipse data constructor built missing
a labelled field
```

Pattern-matching can simultaneously

- distinguish between subtypes
- bind parameter names to values of labeled fields

```
> area :: Shape -> Float
> area (Point {}) = 0
> area (Square {size = s}) = fromInt s ^ 2
> area (Ellipse {size = a, eccentricity = e})
>   = pi * fromInt a^2 * sqrt(1 - e^2)
```

```
Main> area pt
0.0
```

```
Main> area sq
625.0
```

```
Main> area e1
307.812
```

If the same labeled field appears in all of an algebraic type's subtypes, a function can access it without pattern-matching:

```
> setLoc :: Shape -> (Int,Int) -> Shape
> setLoc s xy = s { loc = xy }
```

(This is *not* assignment: setLoc s xy creates a *new* Shape identical to s except that its loc is xy.)

```
> e1' :: Shape
> e1' = setLoc e1 (40, 50)
Main> loc e1'
(40,50)
Main> area e1'
307.812
```

In Pascal and C/C++, the same flexibility can be obtained using *variant records* (in C they're called *unions*), which permit the number and kind of fields to differ among records of the same type:

```
type ShapeKind = (Point, Square, Ellipse)
type Shape = record
    case kind : ShapeKind of
        Point:
            Square: (size : int)
            Ellipse: (size : int, eccentricity: real)
    end;
    locX, locY : int
end
```

The discriminator field is supposed to be tested in code that refers to variant fields:

```
function area( s : Shape ) : real
begin
    case s.kind of
        Point: area := 0.0;
        Square: area := real(s.size)^2
        Ellipse: area := pi * real(s.size)^2
                * sqrt(1 - s.eccentricity^2)
    end
end
```

but the matching is not checked by the compiler.

It is obvious that this flexibility ... gives rise to programming errors that are difficult to detect. In particular, it is now possible to assume in some part of a program that a variable is of a certain variant, whereas it actually is of another variant. This facility is therefore to be used with great caution. —N.Wirth.

Haskell's sum-of-product types have no such loop-hole: The case identifiers are not mere data values, but constructor functions whose arguments' types are part of their definitions.

(The concepts of algebraic types and pattern-matching have been ported into Pizza, an experimental superset of Java.)

### Recursive Algebraic Types

Example: An *expression* is one of

- a literal integer
- a combination of two *expressions* with an operator (+ or -)

This can be modeled by a *recursive* algebraic type:

```
> data Expr = Lit Int
>           | Add Expr Expr
>           | Sub Expr Expr
```

Examples:

```
2          Lit 2
2+3       Add (Lit 2) (Lit 3)
(3-1)+3   Add (Sub (Lit 3) (Lit 1)) (Lit 3)
```

Some functions on expressions

- evaluate
- show (i.e., produce a String representation)

An *evaluate* function:

```
> eval :: Expr -> Int
> eval (Lit n) = n
> eval (Add e1 e2) = eval e1 + eval e2
> eval (Sub e1 e2) = eval e1 - eval e2
```

A *show* function:

```
> showExpr :: Expr -> String
> showExpr (Lit n) = show n
> showExpr (Add e1 e2)
>   = "(" ++ showExpr e1 ++
>     "+" ++ showExpr e2 ++ ")"
> showExpr (Sub e1 e2)
>   = "(" ++ showExpr e1 ++
>     "-" ++ showExpr e2 ++ ")"
```

Are recursive types something new?

Not really— lists are a recursive algebraic type. If they weren't "built-in", we could invent them:

```
> data IntList = Empty | Cons Int IntList
```

Like the built-in lists, algebraic types can be polymorphic:

```
> data List a = Empty | Cons a (List a)
```

Example: Binary Trees

```
> data Tree a = Nil | Node a (Tree a) (Tree a)
```

Some familiar functions for polymorphic Trees:

```
> countTree, depth :: Tree a -> Int
> countTree Nil = 0
> countTree (Node _ t1 t2)
>   = 1 + countTree t1 + countTree t2

> depth Nil = 0
> depth (Node _ t1 t2)
>   = 1 + max (depth t1) (depth t2)
```

Some Tree functions work only for certain types of trees:

```
> occurs :: Tree String -> String -> Int
> occurs Nil p = 0
> occurs (Node n t1 t2) p
>   = oneIf (n==p) + occurs t1 p + occurs t2 p

> sumTree :: Tree Int -> Int
> sumTree Nil = 0
> sumTree (Node n t1 t2)
>   = n + sumTree t1 + sumTree t2

> oneIf :: Num a => Bool -> a
> oneIf True = 1
> oneIf _ = 0
```

### Recursive product types in Pascal and C

How can a Pascal or C programmer define a recursive type?

Suppose we want a Pascal type analogous to the Haskell type

```
> data Expr = Lit Int
>           | Add Expr Expr
>           | Sub Expr Expr
```

Our first attempt uses *variant records*.

```
type ExpKind = (Lit, Add, Sub)
```

```
type Expr = record
```

```
  case kind: ExpKind of
```

```
    Lit: (value: integer)
```

```
    Add: (op1,op2: Expr)
```

```
    Sub: (op1,op2: Expr)
```

```
  end
```

```
end
```

- Every record of type *Expr* contains a record of type *Expr*. Hence its size is *infinite*.

A C example:

```
struct tree { int label; tree left; tree right; };
```

Every tree contains two trees, and hence is infinite.