

COMP 321 Lecture 13: Sum Types

Dan Licata

Fall 2026

This week, we will continue building up the types of our programming language. For now, we will leave out all of the memory-based features of IMP and consider the setting of a simple functional language FUN.

A FUN program consists of a series of top-level function declarations followed by an expression e , and running a program means stepping the expression e until it is done. We will also use the word *value* for a done expression. We will start with the expressions consisting of

- variables x
- numbers k and addition $e_1 + e_2$
- let-bindings of immutable variables $\text{let } x = e_1 \text{ in } e_2$
- binary product types $T_1 \times T_2$ with values written (e_1, e_2) and projections $\text{first } e$ and $\text{second } e$.
- the unit type unit with exactly one value $()$.
- record types $\{f_1 : T_1, \dots, f_n : T_n\}$ with values written $\{f_1 = e_1, \dots, f_n = e_n\}$ and projections $e.f$.

See the Lecture 10/11/12 materials for the typing rules and operational semantics for these.

We will also use top-level functions whose bodies are expressions, written in concrete syntax as $f(x : T_1) : T_2 = e$, with function calls written as $f \ e$ where f is a function name and e is an expression:

$$\frac{x : \tau_1 \vdash e : \tau_2}{(f(x : \tau_1) : \tau_2 = e) \text{ ok}} \quad \frac{(f(x : \tau_1) : \tau_2 = e) \in p \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash f(e) : \tau_2}$$
$$\frac{e \mapsto e'}{f e \mapsto f e'} \quad \frac{(f(x : \tau_1) : \tau_2 = e_2) \in p}{f e_1 \mapsto e_2[e_1/x]}$$

Today, we will work towards adding datatypes to FUN.

1 Motivating Example

Here is an example you might have seen in COMP 212. Suppose you have an application that has several different phases, for example: the user arrives at a web site and is asked to enter their name; the user has entered their name and is asked to enter their password; the user has successfully entered their password and is logged in to an online shopping site. In the first phase, the program has no data about the user yet; in the second phase, the program knows the name (a string); in the third phase, the program knows the name and a current shopping cart (an item list). Each phase can be represented by a datatype constructor that stores the data known to the program in that phase:

```
datatype phase =
  EnterName of unit
| EnterPassword of string
| LoggedIn of string * item list
```

Then the logic of the program can be represented as a transition function between phases, given some user input:

```
fun next(p : phase, input : string) : phase =
  case p of
    EnterName() => EnterPassword(input)
  | EnterPassword(name) => if checkPassword(name, input)
                           then LoggedIn(name, getData(name))
                           else EnterName()
  | LoggedIn(name, cart) => ...
```

2 Enum representation

Some languages like C do not have a built-in notion of datatype. However, using product/record types, we could use an `int` with values 0, 1, or 2 to represent the three phases (EnterName vs. EnterPassword vs. LoggedIn), and then tuple that with the data for *all* of the phases:

```
phase = {which : int, name : string, cart : item list}
```

The datatype constructors are then represented as follows:

```
EnterName() = {which = 0, name = "", cart = []}
EnterPassword(n) = {which = 1, name = n, cart = []}
LoggedIn(n, c) = {which = 2, name = n, cart = c}
```

The `which` field is typically called an *enum* for *enumeration*.

The above `next` function can be written like this:

```
fun next(p : phase, input : string) : phase =
  if p.which = 0 then EnterPassword(input)
  else if p.which = 1 if checkPassword(p.name, input)
                      then LoggedIn(p.name, getData(p.name))
                      else EnterName()
  else if p.which = 2 then ...
  else ...
```

There are a couple of problems with this representation:

- The type `{which : int, name : string, cart : item list}` has more possible values than the datatype we started with. For example, `{which = 3, ...}` does not correspond to any of the three datatype constructors.

- In the program, all of the data for all of the constructors is available in *every* phase, when it should only exist in certain phases. For example, we could accidentally write `p.name` and `p.cart` in the `p.which = 0` case, in which case we would be referring to the “junk” default values that the `EnterName()` constructor initializes these fields to.¹
- In the above example, there are reasonable default values (the empty string, the empty list) to fill in when a piece of data is not present in one of the three phases, but this will not always be the case.

Conceptually, this shows that the `{which : int, name : string, cart : item list}` is big enough to represent all of the datatypes’ values, but it is also strictly *too big* a type — it has more values than we want. Rather than using a product/record type, we can use a *sum type* to more accurately represent what we want.

3 Sum types

We now extend FUN with a *binary sum type* written $\tau_1 + \tau_2$. Here τ_1 and τ_2 are placeholders (metavariables) that stand for any type we want. So this means we have specific binary sum types like `int + int`, `int + string`, `int + (int + int)`, `(int x int) + int` for any two types we choose. It’s useful to give the typing rules and operational semantics for a generic sum type, because then they can be instantiated to any particular type we want.

A binary sum type is like a datatype

```
datatype T1 + T2 =
  left of T1
| right of T2
```

so it has two constructors

```
left  : T1 -> T1 + T2
right : T2 -> T1 + T2
```

The new concrete syntax for these is `left(e)` and `right(e)`. The typing for these is

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{Left}(e) : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{Right}(e) : \tau_1 + \tau_2}$$

In addition, one can case-analyze a value of sum type to see if it was constructed with `left` or `right`, and in each case you get the data stored in that constructor. In concrete syntax, we write

```
case e of left(x) => e1 | right(y) => e2
```

where `x` is a new immutable variable that is in scope in `e1` and `y` is a new immutable variable that is in scope in `e2`. The abstract syntax is `Case(e, x.e1, y.e2)`, which notates that the data of a Case is `e` and `e1` and `e2`, where `e1` has `x` in scope and `e2` has `y` in scope.

¹In C, we could use a union type instead of a product for the data, but we haven’t talked about those yet, and we would still have the problem that the type system would not help correlate the value of the enum with the type of the data available in that case.

The typing rule says that both branches must have the same type, and that x and y must have the type of the data that comes with Left and Right, respectively:

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, x : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{Case}(e, x.e_1, y.e_2) : \tau}$$

The most important cases of the operational semantics say that if we case-analyze a constructed value, we step to the indicated branch, substituting the actual data in the constructor for the variable that stood for such data:

$$\frac{\text{Left}(e) \text{ done}}{\text{Case}(\text{Left}(e), x.e_1, y.e_2) \mapsto e_1[e/x]} \text{ case-instr-left} \quad \frac{\text{Right}(e) \text{ done}}{\text{Case}(\text{Right}(e), x.e_1, y.e_2) \mapsto e_2[e/x]} \text{ case-instr-right}$$

We also have search rules for each:

$$\begin{array}{c} \frac{e \mapsto e'}{\text{Left}(e) \mapsto \text{Left}(e')} \text{ Left-search} \quad \frac{e \text{ done}}{\text{Left}(e) \text{ done}} \\ \frac{e \mapsto e'}{\text{Right}(e) \mapsto \text{Right}(e')} \text{ Right-search} \quad \frac{e \text{ done}}{\text{Right}(e) \text{ done}} \\ \frac{e \mapsto e'}{\text{Case}(e, x.e_1, y.e_2) \mapsto \text{Case}(e', x.e_1, y.e_2)} \text{ Case-search} \end{array}$$

4 Representing Datatypes

As a first example, we can define booleans as

```
bool = unit + unit
true = left()
false = right()
if e then e1 else e2 = case(e, _.e1, _.e2)
```

These definitions satisfy the typing and operational semantics rules from Lecture 10. (In general, *defining* a type means to treat each abstract syntax tree constructor for that type as a meta-level function defined using the other types of the language.)

The phase type from the beginning of class is a little trickier. If the phases were just `EnterName()` and `EnterPassword(name)`, we could use the binary sum type `unit + string`, which is the sum of the data for `EnterName(unit)` and the data for `EnterPassword(a string)`. Then we would represent `EnterName` by `left` and `EnterPassword` by `right`. However, since there are *three* datatype constructors instead of just two, we need to *iterate* the binary sum type twice to represent a ternary sum type. So we define

```
phase = unit + (string + (string x item list))
EnterName() = left()
EnterPassword(name) = right(left(name))
LoggedIn(name, cart) = right(right(name, cart))
```

The next function is mostly the same as before, except we need one `case` for each level of the encoding of the ternary sum:

```

next(p : phase, input : string) : phase =
  case p of
    left(_) => EnterPassword(input)
  | right(q : string + (string x item list)) =>
    case q of
      left(name) => if checkPassword(name, input)
                     then LoggedIn(name, getData(name))
                     else EnterName()
    | right(nc) => ...

```

Like in our original datatype version, the data for each constructor is only available in the case for that constructor, and we do not need an extra case for an unknown `which`.

5 n -ary Labeled Sums and Recursive Datatypes

Any non-recursive datatype

```
datatype D = c1 of T1 | c2 of T2 | ... | cn of Tn
```

can be faithfully represented by the iterated sum type $T_1 + (T_2 + \dots + T_n)$. However, the constructors will be encoded as `Left(v1)`, `Right(Left(v2))`, `Right(Right(Left(v3)))` etc. A more user-friendly syntax is to make an n -ary *labeled* sum type a part of the language:

$$\frac{\Gamma \vdash e_i : \tau_i \quad (\text{datatype } D = \dots \mid c_i \text{ of } \tau_i \mid \dots) \in p}{\Gamma \vdash c_i(e_i) : D}$$

$$\frac{\Gamma \vdash e : D \quad (\text{datatype } D = \dots \mid c_i \text{ of } \tau_i \mid \dots) \in p \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \dots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \text{Case}(e, x_1.e_1, \dots, x_n.e_n) : \tau}$$

If we allow the datatype name D to also occur in the types of the data for the constructors τ_i , we get a *recursive datatype*, such as

```
datatype tree = Empty of unit | Leaf of int | Node of (tree x tree)
```

In general, such a datatype is modeled by a *recursive sum of products*: the datatype is recursive, the different constructors correspond to a sum type, and a single constructor can without loss of generality have exactly one input, because 0 or more than one input can be bundled into a single input using product types.