COMP3161/COMP9164 Supplementary Lecture Notes

Data Types

Liam O'Connor Gabriele Keller Johannes Åman Pohjola

October 23, 2024

1 Composite Data Types

Up to now, we only discussed primitive data types, like integers boolean values, and function types. This is not only an inconvenience, but it seriously restricts the expressiveness of the language. For example, we cannot define a function which returns multiple values at once, or, depending on the input, a boolean or an integer value. In this section we will extend MinHS to have a type system that is able to encode more complex and interesting data types.

1.1 Products

Many languages include ways to group types together into one composite type - a `struct` in C, a `class` in Java. In Haskell, we can bundle a fixed number of values of possibly different types by simply using tuples. For example, the function

```
foo :: Int -> (Bool, Int )
foo x = (even x, x * x )
```

takes an int-value as argument and return a *pair* of a boolean and an integer value. In this example, we can see that `( , )` is overloaded to be both a type constructor (in the type annotation, constructing the type pair of Bool and Int) and a data constructor (in the function body, constructing a new value by bundling a boolean value and an int-value). Similarly, we can construct tuples of (theoretically) arbitrary size in Haskell.[1] More values can be bundled, if this is for some reason necessary, by using nested tuples. There are also ways to define record types, similar to C-structs in that the fields can be referred to by names, not just their position.

In MinHs, we keep it as simple as possible, and only introduce a pair constructor. The type of pairs is called a *product type*, for reasons that will become clear later[2]. The type of the pair of τ_1 and τ_2 is therefore written $\tau_1 \times \tau_2$, in contrast to Haskell.

We also introduce a data constructor, `( , )` to MinHS, which produces values of a product type:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1,\ e_2) : \tau_1 \times \tau_2}$$

Note that we can combine more than two types simply by using nested products: a type `Int` $\times$ (`Int` $\times$ `Int`) bundling three integer values can be constructed with $(3, (4, 5))$.

[1] In practice, most compilers only support tuples up to some fixed size, but according to the standard, it should not be less than 15.

[2] Product types are analogous to cartesian products in set theory, for those that are familiar with sets.

Once we have a pair value, how do we get each component out of it? We introduce a bit more syntax for two *destructors*, `fst` and `snd`, which, given a pair, return the first or second component of the pair respectively:

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ e : \tau_2}$$

Evaluation rules are omitted as they are straightforward.

1.2 Sums

Sum types are less common than product types in programming languages, but they can be simulated with tagged `union`s in C and interface inheritance in Java. In essence $\tau_1 + \tau_2$ allows *either* a value of type τ_1 or a value of type τ_2, but not both. In Haskell, an equivalent definition would be:[3]

```
data Sum a b = InL a | InR b
```

We will introduce similar constructors, `InL` and `InR`, for our sum type in MinHS:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{InL}\ e : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{InR}\ e : \tau_1 + \tau_2}$$

Once again, we can combine multiple types to give n-ary sums via nesting.

Instead of function-like destructors for sum types, we will introduce a new `Case` expression, similar to Haskell's equivalent, which provides two alternative expressions. One for `InL`, and one for `InR`. As an example:

```
recfun sumToProd :: (Int + Int -> Bool * Int) e
    = case e of InL u  = (False, u)
                InR v  = (True,  v);
```

The abstract syntax of a case expression is as follows: $\texttt{Case}\ \tau_1\ \tau_2\ e\ (x.e_1)\ (y.e_2)$, where the input sum value e is of type $\tau_1 + \tau_2$, and e_1 is the alternative for `InL`, and e_2 is the alternative for `InR`. Typing rules:

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

Big step evaluation rules:

$$\frac{e \Downarrow \texttt{InL}\ v \quad e_1[x := v] \Downarrow r}{\texttt{Case}\ \tau_1\ \tau_2\ e\ (x.e_1)\ (y.e_2) \Downarrow r} \quad \frac{e \Downarrow \texttt{InR}\ v \quad e_2[y := v] \Downarrow r}{\texttt{Case}\ \tau_1\ \tau_2\ e\ (x.e_1)\ (y.e_2) \Downarrow r}$$

With this, we can also make a four-valued type: $\texttt{Bool} + \texttt{Bool}$ has values `InL False`, `InR False`, `InL True`, `InR True`.

[3] The Haskell standard library calls this type constructor `Either`, and names the data constructors `Left` and `Right`.

1.3 Unit Type

Could we define a type with only three values? At the moment, the smallest type we've got is `Bool`, which has two values, and both products and sums of `Bool` give us a type too large with four values.

The way we resolve this is to introduce a new type, called $\mathbf{1}$, sometimes written $\top$, which has exactly one value - `()`. It can be thought of as the "empty" tuple, and corresponds to a `void` type in C.

Typing rules are obvious:

$$\frac{}{\Gamma \vdash \texttt{()} : \mathbf{1}}$$

With this type, we can construct a type of three values using a sum type: The type $\mathbf{1} + (\mathbf{1} + \mathbf{1})$ has three values: `InL ()`, `InR (InL ())`, `InR (InR ())`.

1.4 Empty Type

We add another type, called $\mathbf{0}$, that has *no* inhabitants. This can be useful in a function's type, as a function like `Int` $\to \mathbf{0}$ indicates that the function will not return[4].

Because it is empty, there is no way to construct it.

We do have a way to eliminate it, however:

$$\frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathsf{absurd}\ e :\ \tau}$$

If I have a variable of the empty type in scope, we must be looking at an expression that will *never* be evaluated. Therefore, we can assign any type we like to this expression, because it will never be executed.

1.5 Type Isomorphism

The above type of three values could be represented as $\mathbf{1} + (\mathbf{1} + \mathbf{1})$ as shown, but it could also be represented as $\mathbf{1} + \texttt{Bool}$ (as it also has three values). We will define the notion of *type isomorphism* to capture the similarity between these terms.

Formally, two mathematical objects A and B are considered *isomorphic* if there exists a structure preserving mapping (or a *morphism*) f from A to B, and an inverse morphism f^{-1} from B to A such that $f \circ f^{-1}$ is an identity morphism I_A from A to A (i.e $f(f^{-1}(x)) = x$) and $f^{-1} \circ f$ is an identity morphism I_B from B to B (i.e $f^{-1}(f(y)) = y$). It is analogous to bijection in set theory or logic, and is often written as $A \simeq B$ where A is isomorphic to B.

Seeing as types can be thought of as (slightly restricted) sets of values, we can construct such mappings to show types A and B isomorphic if and only if A has the same number of values as B.

As an example, we could map `InL ()` to `InL ()`, `InR (InL ())` to `InR False` and `InR (InR ())` to `InR True` (and vice-versa) and thus show that $\mathbf{1} + (\mathbf{1} + \mathbf{1}) \simeq \mathbf{1} + \texttt{Bool}$. Generally, it is sufficient to show that the two types have the same number of values in order to show that they are isomorphic.

As you have probably already realised, computing the size of types can be achieved easily by using this function:

[4]Some functions are expected to not return, for example functions that throw exceptions.

$$
\begin{array}{lcll}
|\mathbf{0}| & = & 0 & \\
|\mathbf{1}| & = & 1 & \\
|\texttt{Bool}| & = & 2 & \\
|\texttt{Int}| & = & 2^{32} & \\
|\tau_1 \times \tau_2| & = & |\tau_1| \times |\tau_2| & \\
|\tau_1 + \tau_2| & = & |\tau_1| + |\tau_2| & \\
|\tau_1 \to \tau_2| & = & |\tau_2|^{|\tau_1|} & \text{for total, terminating functions only}
\end{array}
$$

1.6 Type Algebra

Since type equivalence is based simply on the number of values within the types, and sum and product types correspond to addition and multiplication of type size, types follow all the same algebraic laws as the natural numbers. Or in other words, the types form a *commutative semiring*.

Laws for $(\tau, +, \mathbf{0})$:

- Associativity: $(\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)$
- Identity: $\mathbf{0} + \tau \simeq \tau$
- Commutativity: $\tau_1 + \tau_2 \simeq \tau_2 + \tau_1$

Laws for $(\tau, \times, \mathbf{1})$

- Associativity: $(\tau_1 \times \tau_2) \times \tau_3 \simeq \tau_1 \times (\tau_2 \times \tau_3)$
- Identity: $\mathbf{1} \times \tau \simeq \tau$
- Commutativity: $\tau_1 \times \tau_2 \simeq \tau_2 \times \tau_1$

Combining $\times$ and $+$:

- Distributivity: $\tau_1 \times (\tau_2 + \tau_3) \simeq (\tau_1 \times \tau_2) + (\tau_1 \times \tau_3)$
- Absorption: $\mathbf{0} \times \tau \simeq \mathbf{0}$

1.7 Curry-Howard Correspondence

Before moving on, let us gather all the typing rules we have defined so far, including typing rules for λ-functions and application:

$$
\frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathsf{absurd}\ e : \tau} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}
$$

$$
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{InL}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{InR}\ e : \tau_1 + \tau_2}
$$

$$
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{InL}\ x \to e_1; \mathsf{InR}\ y \to e_2) : \tau}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ e : \tau_2}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}
$$

If I remove all the *terms*, that is, everything between the $\vdash$ and the type, and just leave contexts and types in the rules, I get something quite remarkable:

$$\frac{\Gamma \vdash \mathbf{0}}{\Gamma \vdash \tau} \qquad \frac{}{\Gamma \vdash \mathbf{1}}$$

$$\frac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash \tau_1 + \tau_2 \qquad \tau_1, \Gamma \vdash \tau \qquad \tau_2, \Gamma \vdash \tau}{\Gamma \vdash \tau}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \times \tau_2}{\Gamma \vdash \tau_1} \qquad \frac{\Gamma \vdash \tau_1 \times \tau_2}{\Gamma \vdash \tau_2}$$

$$\frac{\Gamma \vdash \tau_1 \to \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2} \qquad \frac{\tau_1, \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}$$

As it happens, the rules for our types correspond exactly to the rules of constructive propositional logic!

The type **1** corresponds to `True` or $\top$, the type **0** corresponds to `False` or $\bot$, sum types correspond to disjunctions, and product types to conjunctions. Functions correspond to implication.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P} \qquad \frac{}{\Gamma \vdash \top}$$

$$\frac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2} \qquad \frac{\Gamma \vdash P_2}{\Gamma \vdash P_1 \vee P_2}$$

$$\frac{\Gamma \vdash P_1 \vee P_2 \qquad P_1, \Gamma \vdash P \qquad P_2, \Gamma \vdash P}{\Gamma \vdash P}$$

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2} \qquad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_1} \qquad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_2}$$

$$\frac{\Gamma \vdash P_1 \to P_2 \qquad \Gamma \vdash P_1}{\Gamma \vdash P_2} \qquad \frac{P_1, \Gamma \vdash P_2}{\Gamma \vdash P_1 \to P_2}$$

This means that constructing a well-typed program is equivalent to constructing a proof of the proposition encoded by its type. Programs are proofs, and types are propositions.

This correspondence goes by many names, but is usually attributed to Haskell Curry and William Howard. It is a *very deep* result, carrying through every aspect of programs and proving:

Programming	**Logic**
Types	Propositions
Programs	Proofs
Evaluation	Proof Simplification

It turns out, no matter what logic you want to define, there is always a corresponding λ-calculus, and vice versa.

Constructive Logic	Typed λ-Calculus
Classical Logic	Continuations
Modal Logic	Monads
Linear Logic	Linear Types, Session Types
Separation Logic	Region Types

Philip Wadler has written a much more detailed exposition of this correspondence in this very readable paper, located here:

`http://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf`

He has also given a nice talk on the topic at Strange Loop in 2015:

`https://www.youtube.com/watch?v=IOiZatlZtGU`

Some examples of proofs of basic logical properties are given below:

To prove that $A \wedge B \to B \wedge A$, i.e. commutativity of conjunction, we can just flip the order of the values in the pair:

$$\begin{aligned} &\mathit{andComm} :: A \times B \to B \times A \\ &\mathit{andComm}\ p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{aligned}$$

We can prove transitivity of implication as well:

$$\begin{aligned} &\mathit{transitive} :: (A \to B) \to (B \to C) \to (A \to C) \\ &\mathit{transitive}\ f\ g\ x = g\ (f\ x) \end{aligned}$$

Transitivity of implication is just *function composition*.

1.7.1 Caveats

This correspondence is predicated on the condition that all functions we define have to be *total and terminating*. Otherwise we get an *inconsistent* logic that lets us prove false things:

$$\begin{aligned} &\mathit{proof}_1 :: \mathsf{P} = \mathsf{NP} \\ &\mathit{proof}_1 = \mathit{proof}_1 \end{aligned}$$

$$\begin{aligned} &\mathit{proof}_2 :: \mathsf{P} \neq \mathsf{NP} \\ &\mathit{proof}_2 = \mathit{proof}_2 \end{aligned}$$

Most common calculi correspond to constructive logics, not classical ones, so principles like the *law of excluded middle* or double negation elimination do not hold:

$$\neg\neg P \to P$$

It is possible to construct λ-calculi where they do hold, however for the purposes of this course, we will concern ourselves primarily with constructive logics.

2 Recursive Types

What about if we wanted to write a linked list? In Haskell, this would be easily defined:

```
data ListOfInt = Cons Int ListOfInt
               | Nil
```

In MinHS, however, we are stuck - we have no name by which we can introduce recursive types. All types in MinHS are anonymous. What we want to be able to declare is the type $\tau = (\texttt{Int} \times \tau) + \mathbf{1}$ (which is isomorphic to the Haskell definition above).

To solve this, we introduce a new type system feature, called `Rec` (sometimes written μ). It would allow a linked list type to be written as follows:

$$\texttt{Rec}\, t.\, (\texttt{Int} \times t) + \mathbf{1}$$

We also need to introduce constructors and destructors for this `Rec` construct. We call them `Roll` and `Unroll`. The typing rules for `Roll` are straightforward, and mostly uninteresting:

$$\frac{\Gamma \vdash x : \tau[t := \texttt{Rec}\, t.\, \tau]}{\Gamma \vdash \texttt{Roll}\ x : \texttt{Rec}\, t.\, \tau}$$

`Unroll`, however, gives us an insight into how these recursive types actually work:

$$\frac{\Gamma \vdash x : \texttt{Rec}\, t.\, \tau}{\Gamma \vdash \texttt{Unroll}\ x : \tau[t := \texttt{Rec}\, t.\, \tau]}$$

That is, when we `Unroll` a recursive type, the recursive name variable (t) is replaced with *the entire recursive type*. Each `Unroll` eliminates one level of recursion.

We can define a value of our linked list type, written in Haskell as $[3, 4]$, as follows:

$$\texttt{Roll}\ (\texttt{InL}\ (\ (3, (\texttt{Roll}\ (\texttt{InL}\ (\ (4, (\texttt{Roll}\ (\texttt{InR}\ ()))))))))$$

Not very pretty, but isomorphic to the Haskell list.

3 Haskell Datatypes

MinHS's type system, with these extensions, is now nearly as powerful as Haskell's. To show this, I will demonstrate a technique which gives an isomorphic type in MinHS to any (non-polymorphic) type in Haskell. Suppose we have the following Haskell type:

$$\texttt{data Foo} = \texttt{Bar Int Bool} \mid \texttt{Baz} \mid \texttt{Herp Foo}$$

First, replace all data constructors with $\mathbf{1}$:

$$\texttt{data Foo} = \mathbf{1}\ \texttt{Int Bool} \mid \mathbf{1} \mid \mathbf{1}\ \texttt{Foo}$$

Then, multiply the constructors with all of their arguments:

$$\texttt{data Foo} = \mathbf{1} \times \texttt{Int} \times \texttt{Bool} \mid \mathbf{1} \mid \mathbf{1} \times \texttt{Foo}$$

Then, replace all the alternatives with sums:

$$\texttt{data Foo} = \mathbf{1} \times \texttt{Int} \times \texttt{Bool} + \mathbf{1} + \mathbf{1} \times \texttt{Foo}$$

Apply algebraic simplification (usually just $\mathbf{1} \times x \simeq x$):

$$\texttt{data Foo} = \texttt{Int} \times \texttt{Bool} + \mathbf{1} + \texttt{Foo}$$

Then, replace the recursive references (if any) with a `Rec` construct:

$$\texttt{Rec}\, t.\, \texttt{Int} \times \texttt{Bool} + \mathbf{1} + t$$

As can be seen from the above example, this technique will convert non polymorphic, non parameterised Haskell types into isomorphic MinHS ones. A useful trick for your exam ;)