Concepts of Programming Language Design

Data types in Explicitly Typed Languages

Liam O'Connor-Davis, Gabriele Keller

November 6, 2024

1. **Data and Type Constructors**

   (a) Just looking at the dynamic semantics of MinHs with algebraic data types: what is the difference between constructors, such as `Pair`, `Inl`, `Inl`, `Inr`, and `Roll`, and destructors such as `Fst`, `Snd`, `Case`, and `Unroll`?

   > **Solution:** Terms formed by constructors are already in a final state (for strict semantics: if their arguments are fully evaluated), whereas there are evaluation rules for the destructors.

   (b) What are possible types for the following MinHs terms?

   1. `(3, True)`
   2. `snd (3, True)`
   3. `Inl (3, True)`
   4. `Roll (3, True)`
   5. `Roll (Inl (3, True))`

   > **Solution:**
   >
   > 1. `(3, True) :: Int * Bool`
   > 2. `snd (3, True) :: Bool`
   > 3. `Inl (3, True) :: (Int * Bool) + (Int -> Int)`, or any other type of the form `(Int * Bool) + a`, where `a` is replaced with any legal MinHs type.
   > 4. `Roll (3, True) :: Rec t. (Int * Bool)`, where the choice of the variable name `t` is arbitrary. It's a somewhat pointless type, but it's a legal MinHs type.
   > 5. `Roll (Inl (3, True)) :: Rec t. (Int * Bool) + t` is a possible type, but so is `Rec t. (Int * Bool) + Int` or `Rec t. (Int * Bool) + (t + ())`. That is, any type of the form `Rec t. (Int * Bool) + a`, where `a` is a placeholder for a MinHs type which may (but does not have to) contain the type variable `t`.

2. **Relating Haskell and MinHs Types**: Determine a MinHS type that is isomorphic to the following Haskell type declarations:

   (a) `data Maybe Int = Just Int | Nothing`

   > **Solution:** Solutions may vary, but `Int` + `Unit` is the simplest.

   (b) `data Nat = Zero | Suc Nat`

   > **Solution:** `Rec` t. `Unit` + t

   (c) `data IntTree = Tree Int IntTree IntTree | Leaf Int`

> **Solution:** $\texttt{Rec}\, t.\, (\texttt{Int} * t * t) + \texttt{Int}$

3. **Inhabitation**: Do the following MinHS types contain any (finite) values? If not, explain why. If so, give an example value. For those who do not, can you write a function which, according to the typing rules given in the lecture, can have this type as return type?

   (a) $\texttt{Rec}\, t.\, \texttt{Int} + t$

   > **Solution:** Yes, `(Roll (Inr (Roll (Inl 3))))` is an example finite value.

   (b) $\texttt{Rec}\, t.\, \texttt{Int} * t$

   > **Solution:** No, there is no finitie value. The only way to express a value of this type is to write a recursive definition
   >
   > ```
   > let x = Roll  (5, x)
   > in ...
   > ```
   >
   > which is not allowed in MinHs, since the scope of x is just the body of the let-binding, not the right hand side of the binding.

   (c) $(\texttt{Rec}\, t.\, \texttt{Int} * t) + \texttt{Bool}$

   > **Solution:** Yes, the only finite values are `(Inr True)` and `(Inr False)`. All other values are infinite.

4. **Isomorphism**

   (a) What types (other than `Bool`) are isomorphic to `Bool`? Give an example, and show how it is isomorphic by providing the mapping from `Bool` and inverse mapping to `Bool`.

   > **Solution:** The simplest is $\tau = \texttt{Unit} + \texttt{Unit}$, where `True` is isomorphic to `(Inl unitel)` and `False` is isomorphic to `(Inr unitel)` (or the other way around).

   (b) What types (other than `Int`) are isomorphic to `Int`? Give an example and show it to be isomorphic. Remember that `Int` in MinHS is the full set $\mathbb{Z}$ not just machine integers.

   > **Solution:** One of the simplest is $\tau = (\texttt{Rec}\, t.\, \texttt{Unit} + t) * \texttt{Bool}$. The mapping works as follows:
   >
   > $$f(x) = \begin{cases} x < 0, & (\texttt{pair}\ g(-x-1), \texttt{False}) \\ x >= 0, & (\texttt{pair}\ g(x), \texttt{True}) \end{cases}$$
   >
   > $$\text{Where } g(x) = \begin{cases} x = 0, & (\texttt{Roll (Inl unitel)}) \\ x > 0, & (\texttt{Roll (Inr}\ g(x-1)\texttt{))} \end{cases}$$
   >
   > The inverse mapping is very similar:
   >
   > $$f(x) = \begin{cases} x = (\texttt{pair}\ n, \texttt{True}\ ), & g^{-1}(n) \\ x = (\texttt{pair}\ n, \texttt{False}), & -g^{-1}(n) - 1 \end{cases}$$
   >
   > $$\text{Where } g^{-1}(x) = \begin{cases} x = (\texttt{Roll (Inl unitel)}), & 0 \\ x = (\texttt{Roll (Inr}\ x'\texttt{))}, & 1 + g^{-1}(x') \end{cases}$$
   >
   > As $f \circ f^{-1} = I_{\mathbb{Z}}$ and $f^{-1} \circ f = I_\tau$, τ is isomorphic to `Int`.

5. **Functional Programming**: A type isomorphic to a list of integers in MinHS is simply $\texttt{Rec}\, t.\, ((\texttt{Int} * t) + Unit)$. Implement a variety of utility functions to manipulate these lists in MinHS.

   (a) The function *sum*, which computes the sum of a list of integers.

   > **Solution:**

```
recfun sum :: ((Rec t. ((Int * t) + Unit)) -> Int) x =
    case (Unroll x) of
      Inl l -> fst l + sum (snd l)
      Inr e -> 0
```

(b) The function *map*, which takes a function f of type `Int` → `Int` and an input list i, returning a new list which consists of f applied to every element of the list i.

Solution:

```
recfun map :: ((Int -> Int) * (Rec t. ((Int * t) + Unit))
           -> (Rec t. ((Int * t) + Unit))) x =
    case (Unroll (snd x)) of
      Inl l -> Roll(Inl (fst x (fst l), map (fst x, snd l)))
      Inr e -> Roll(Inr e)
```

(c) The function *filter*, which takes a predicate p of type `Int` → `Bool`, and an input list i, returning a new list which consists of all elements x in i for which $p(x)$ is `True`.

Solution:

```
recfun filter :: ((Int -> Bool) * (Rec t. ((Int * t) + Unit))
              -> (Rec t. ((Int * t) + Unit))) x =
    case (Unroll (snd x)) of
      Inl l -> if (fst x (fst l))
          then Roll(Inl (fst l, filter (fst x, snd l)))
          else filter (fst x, snd l)
      Inr e -> Roll(Inr e)
```

(d) The function *foldl*, which takes an initial value (called an accumulator) a (of type Int), a binary function op : `Int` ∗ `Int` → `Int`, and a list of integers. When given an empty list, the function returns the accumulator. For a non-empty list with head h and tail t, it will recursively call itself with the new accumulator being $op(a, h)$, the same op, and the list t.

Solution:

```
recfun foldl :: (Int * (Int * Int -> Int) * (Rec t. ((Int * t) + Unit)) -> Int)
        args =
    let a = fst args in
    let op = fst (snd args) in
    let list = snd (snd args) in
      case (Unroll list) of
        Inl l -> foldl (op(a,fst l),(op, snd l))
        Inr e -> a
```

(e) Implement *sum* and also *product* (which multiplies a list of integers) using *foldl*.

Solution:

```
recfun sum :: ((Rec t. ((Int * t) + Unit)) -> Int) list =
    foldl (0, (recfun add :: (Int * Int -> Int) xy = fst xy + snd xy), list)

recfun product :: ((Rec t. ((Int * t) + Unit)) -> Int) list =
    foldl (1, (recfun mul :: (Int * Int -> Int) xy = fst xy * snd xy), list)
```