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`?
   (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))`

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`
   (b) `data Nat = Zero | Suc Nat`
   (c) `data IntTree = Tree Int IntTree IntTree | Leaf 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) $\texttt{Rec}\, t.\, \texttt{Int} + t$
   (b) $\texttt{Rec}\, t.\, \texttt{Int} * t$
   (c) $(\texttt{Rec}\, t.\, \texttt{Int} * t) + \texttt{Bool}$

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`.
   (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.

5. **Functional Programming**: A type isomorphic to a list of integers in MinHS is simply $\texttt{Rec}\, t.\, ((\texttt{Int} * t) + \mathit{Unit})$. 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.
   (b) The function *map*, which takes a function f of type $\texttt{Int} \to \texttt{Int}$ and an input list i, returning a new list which consists of f applied to every element of the list i.
   (c) The function *filter*, which takes a predicate p of type $\texttt{Int} \to \texttt{Bool}$, and an input list i, returning a new list which consists of all elements x in i for which $p(x)$ is `True`.
   (d) The function *foldl*, which takes an initial value (called an accumulator) a (of type Int), a binary function $\mathit{op} : \texttt{Int} * \texttt{Int} \to \texttt{Int}$, 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 $\mathit{op}(a, h)$, the same op, and the list t.
   (e) Implement *sum* and also *product* (which multiplies a list of integers) using *foldl*.