1 A Concrete Implementation

We now give a first implementation of these constructs. Our first goal is to write a heap and a reference that allow us to manipulate the heap in a stateful fashion. Since our heap is strongly typed, our first need is a collection of heterogeneous values where values may be added to the list; this kind of collection can be easily implemented with heterogeneous lists accessed with compile-time Church Numerals. Our implementation is based on that of ...

Note that we never use "regular" lists in our system. For this reason we have decided to borrow the notation where :: is the list constructor since we will not need it elsewhere. Also, in this context we will use the same symbols twice: once for values and once for types; we advise the reader to be always aware of the context in which these operators are used since the context will make it clear which one of the symbols is currently in use. We believe that having two distinct sets of symbols in the end would reduce the readability of our code and so we only used distinct symbols in the actual implementation.

A heterogeneous list can be built using two type constructors, one for the empty heterogeneous list and one for adding a new head to an existing heterogeneous list:

```
Nil=Nil
h::tl=h::tl
```

We characterize heterogeneous lists not by their type (that would be impossible since heterogeneous lists all have very different types), but rather with the $HList$inductive predicate:

```
HList Nil
HList tl ⇒ HList (h::tl)
```

This means that the empty heterogeneous list is a heterogeneous list, and by adding an element of any type h to a heterogeneous list we obtain a new heterogeneous list.

Heterogeneous lists are accessed by index. To make accesses type safe we need to ensure that no integer that would result in an impossible access (index out of range) is ever used to access a list. For this reason we implement compile-time Church Numerals at the type level, and we use those to access our heterogeneous lists:

```
Z=Z
S n=S n
```

Just like we did for heterogeneous lists we now add a $CNum$predicate to characterize all Church Numerals inductively:

```
CNum Z
CNum n ⇒ CNum (S n)
```

The only real difference between Church Numerals and heterogeneous lists is that while the heterogeneous list carries inside its values a series of values of different types, a Church Numeral is always empty, that is beyond a certain number of applications of its S constructor it does nothing. For this reason we will never be interested in the actual value of Church Numerals, and we will only use unnamed instances of Church Numerals like in the following example:

```
CNum n ⇒ f : n → Nil
f (_:n)=Nil
```

Accessing a heterogeneous list is based on the *HLookup* predicate:

```
CNum n ⇒ HLookup l n
```

This predicate contains a type function that tells us the type of the selected field:

```
HAt l n : *
```

Plus two functions to read and write the n-th element of the list:

```
hRead : l → n → HAt l n
hWrite :HAt l n → l → n → l
```

The *HLookup* predicate is instanced inductively on the second parameter. First we instance *HLookup* for accesses to the first item of the list (index 0):

```
HLookup (h::tl) Z
HAt (h::tl) Z=h
hRead (h::tl) _ = h
hWrite v (h::tl) _ =v::tl
```

Now we instance *HLookup* for all other accesses:

```
CNum n ⇒ HLookup (h::tl) (S n)
HAt (h::tl) (S n)=HAt tl n
hRead (h::tl) _=hRead tl (_:n)
hWrite v (h::tl) _=h::(hWrite v tl (_:n))
```

Notice two interesting aspects about this code: first it is impossible to read anything on the empty heterogeneous list; second, it is impossible to access a heterogeneous list with an index that would be too big. As an example, consider the type of the expression:

```
hRead (0::1::Nil)(S S S Z) : α
```

From the definition of the *hRead* function we know that:

$$\begin{aligned}\alpha &= \text{HAt } (\mathbf{Int}::\mathbf{Int}::\text{Nil}) \ (\text{S S S Z})\\ &= \text{HAt } (\mathbf{Int}:\text{Nil})(\text{S S Z})\\ &= \text{HAt Nil } (\text{S Z})\end{aligned}$$

But the last term is stuck since there are no instances of the HAttype functions (being there no instances of the $HLookup$ predicate) where$l = Nil$.

A straightforward choice for a reference type is clearly inspired by the state monad: a reference will contain a pair of functions, one for evaluating the reference from the heap and one for setting a new value for this reference on the heap:

```
Reference h α = Reference (h → (α × h)) (α → h → (Unit × h))
```

The reference operations are stateful, that is the heap might be changed (its new value is returned by both the get and the set functions) even when we simply evaluate the reference. These two functions have been chosen for the simplicity with which they can be converted into statements of the state monad.

Now that we have both a candidate for heaps and references we can instance the $Heap$ predicate. We will use heterogeneous lists as typed heaps and $Reference$ as the type for our references. We start by inductively instancing the allocation operators:

```
Heap Nil Reference
  New Nil α = α :: Nil
  new Nil x = x :: Nil
  delete (x::Nil) = Nil
```

and

```
Heap (h::tl) Reference
  New (h::tl)α =h::(New α tl)new (h::tl)x=h::(new x tl)delete (h::tl)=h::(delete tl) \]
```

The remaining functions needed for one heap have the same implementation for both the empty heap and a non-empty heap. For this reason we just assume that $HListh \Rightarrow Heaph$.

The evaluation operator simply returns the get function:

```
eval (Reference get set)=ST get
```

The assignment operator invokes the set function with the given parameter and returns the resulting curried function:

```
(Reference get set) := v = ST(set v)
```

The allocation operator ($\gg +$) generates a temporary reference that refers to a newly allocated value in a larger heap. A function is invoked to act on this reference to generate a statement that will return the final value of the computation done on this new reference. We define this operator by steps; since the ($\gg +$) operator requires manipulating the length of a heterogeneous list, we begin by giving the *HLength* type function (which is part of the *HList* type predicate):

```
HLength l : *
HLength Nil=Z
HLength (h::tl)=S (HLength tl)
```

The first step of the actual computation is to build the new reference to access the new item, which can be looked up with an index equal to the length of the input heap:

```
(>>+) : α → (Reference (New h α) α → ST (New h α) β) → ST h β
 v >>+ k =
    let r_new = Reference (λh.(hRead h (_:HLength h),h))
                          (λv.λh.((),hWrite v h (_:HLength h)))
```

Notice how we invoke the read and write functions (*hRead* and *hWrite*) with index equals to the length of the type of the original heap h. The second step consists in obtaining the result by invoking the k function:

```
    let (ST res)=k r_new
```

Finally we convert this result into the desired form by temporarily adding the new value v to the heap, performing the computation and then removing the value:

```
    in ST(λh.downcast(res(new h v)) where downcast (x,h)=(x,delete h)
```