Abstract Type of Maps

Robert Harper

October 13, 2017

We will make use of a class of ordered types:

data `order = LSS | EQL | GTR`
signature `ORD =` **sig**
  type `t`
  val `compare : t × t → order`
  val `= : t × t → bool`
  val `≤ : t × t → bool`
end

The class of monoids on a type of generators:

signature `MONOID =` **sig**
  type `g`
  type `t`
  val `e : t`
  val `⋆ : t × t → t`
  val `i : g → t`
end

These are understood to satisfy the monoid laws (associativity and unit laws), with $\star$ as multiplication and `e` as unit element.

We will need the data type

$$\textbf{data}\ \alpha\ \texttt{inf} = -\infty \mid \texttt{i of}\ \alpha \mid \infty,$$

which extends an argument type with points at infinity. If the type `t` is an ordered type, then the type `t inf` may be ordered such that $-\infty$ is smaller than any value $\texttt{i}(\texttt{v})$, all of which values are themselves smaller than ∞, and $-\infty$ is smaller than ∞.

The abstract type of maps from keys to elements, with reduced values, is specified by the following signature:

```
signature MAP = sig
  structure Key : ORD
  type key = Key.t
  type elt
  type entry = key × elt

  structure RVal : MONOID with g=entry
  type rval = RVal.t

  type map

  val emp : map
  val sing : entry → map

  (* keys in left must be strictly smaller than those in right *)
  val join : map × map → map

  (* split at a key, return associated element, present *)
  val split : map → key → map × elt option × map

  type α mon = α × (entry → α) × (α × α → α)
  val mapred : α mon → map → α

  (* generic reduced value *)
  val rval : map → rval

  (* constant-time computations *)
  val size : map → int
  val minkey : map → key inf
  val maxkey : map → key inf
end
```

The type `map` is to be thought of as the free monoid on values of type `entry` as generators. That is to say, we have the following structures:

```
structure Map :> MAP = ...

structure MapAsMonoid : MONOID = struct
  type g = Map.entry
  type t = Map.map
```

```
  val e = Map.emp
  val i = Map.sing
  val * = Map.join
end
```

Moreover, `Map.emp` and `Map.join` satisfy the monoid laws. Consequently, the first argument to `mapred` must itself satisfy the monoid laws in order for the result to be well-defined.

The semantics of `mapred(e,i,*)` is specified by the equations in the context of an open `Map` structure:

```
mapred (e,i,*) emp = e
mapred (e,i,*) (sing p) = i(p)
mapred (e,i,*) (join (m1, m2)) =
  (mapred (e,i,*) m1) * (mapred (e,i,*) m2)
```

In particular we may define `size` to be

```
mapred (0,const 1,+),
```

which computes the number of entries in a map. Similarly, we may define `minkey` to be

```
mapred (∞,λ(k,_)⇒k,max)
```

and `maxkey` to be

```
mapred(−∞,λ(k,_)⇒k,min).
```

Here `min` and `max` are to be taken in the sense of the extended ordering with points at infinity.

The `mapred` operation may be used to implement

```
filter : (entry → bool) → map → map
```

as folows:

```
filter p =
  mapred (emp, λx ⇒ if p x then sing x else emp, join).
```

That is, each entry is replaced by either the empty or the singleton map, according to whether the predicate holds of it or not, and these are joined to obtain the filtered map.

The `split` operation behaves according to the following equations:

```
split (emp, k) = (emp, Nothing, emp)

split (sing (k,v), k') =
  if k=k' then
    (emp, Just v, emp)
```

```
    else if k<k' then
      (sing (k,v), Nothing, emp)
    else
      (emp, Nothing, sing (k,v))

  split (join (m1,m2),k) =
    if k ≤ maxkey m1 then
      let (m11, o, m12) = split (m1, k) in (m11, o, join (m12, m2))
    else
      let (m21, o, m22) = split (m2, k) in (join (m1, m21), o, m22)
```

We may define

```
find : Map.map * Map.key → Map.elt option
```

in terms of split as follows:

```
find(m,k) = let (_, o, _) = split (m, k) in o.
```

As an exercise, you may use the foregoing equations governing `split` to derive equations that specify the behavior of `find` to be as expected.

Reduced values are a way to maintain the result of `mapred` for a particular monoid during the construction of the map so that the result may be computed in constant time. The key equation is that if the reduced value computation is given by `(e,i,*)`, in such a way that it obeys the monoid laws, then

```
rval m = mapred (e,i,*) m
```

That is, the reduced value is the reduction of the map according to the specified monoid! For example, we may maintain the size of a map as a reduced value using the monoid structure specifed earlier so that it may be determined in constant time. Similarly, the largest and smallest keys in a map may be maintained as reduced values using the same method.

Reduced values are implemented using *augmentation*: the underlying tree structure is enriched with an additional construct that holds the augmented values associated with each tree. In the present case we associate a generic augmented value of type `rval`, as well as the size of type `int` and the minimum and maximum keys, both of type `key inf`.

```
data tree = Tree of bst * rval * int * key inf * key inf
and  bst =  Empty | Node of tree * entry * tree
```

Notice that the augmented values are intertwined among the nodes of the binary search tree, and surround each empty binary search tree. The "entry point" is the type `tree`; the type `bst` is an "auxiliary" used to represent the internal structure of the tree.

The augmented values may be maintained using *smart constructors*, which create empty and non-empty trees, respectively.

```
val empty = Tree of (Empty, RVal.e, 0, ∞, −∞)
val node =
λ(lt as Tree (lb, lr, ls, li, la), kv,
  rt as Tree (rb, rr, rs, ri, ra)) ⇒
 Tree
  (Node (lb, kv, rb),
   RVal.⋆ (lr, rr),
   ls+rs,
   min (li, ri),
   max (la, ra))
```

The smart constructors, `empty` and `node`, maintain the reduced values according to their definitions in terms of `mapred`.

Finally, it often helps to structure the implementation using an `expose` operation of type `tree→bst` that "exposes" the underlying structure of the tree for pattern-matching purposes. Doing so provides only one level deep of pattern matching, but this is sufficient for nearly all situations.