Under consideration for publication in J. Functional Programming

Data structures as closures

Greg Michaelson and Robert Stewart
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, Scotland
(*e-mail:* G.Michaelson@hw.ac.uk; R.Stewart@hw.ac.uk)

Abstract

In formalising denotational semantics, Strachey introduced a higher order update function for the modelling of stores, states and environments. This function relies solely on atomic equality types, λ abstractions and conditions to represent stack disciplined association sequences as structured closures, without recourse to data structure constructs like lists.

Here, we present higher order functions that structure closures to model queue, linear ordered and tree disciplined key/value look up functions, built from a subset of OCaml equivalent to moderately sugared pure λ functions. We also discuss their type and performance properties.

1 Introduction

For many problems, there is a need to store pairs of keys and values, for access by key. Typical real-world key/value APIs are implemented using data structures, for example the Haskell `Data.Map` module in the `containers` library is implemented using efficient size balanced tree structures (Adams, 1993). Usually, update and search are separate activities.

For illustration, we'll use OCaml. In what follows, we will use $\Rightarrow ... \Rightarrow$ to indicate a judicious mix of α renaming, and β and η reduction. To implement a key/value map, we will assume a list data structure corresponding to:

```
(* list data structure supporting a key/value API *)
type 'a list = Nil | Cons of 'a * 'a list;;
```

and will use `::` as an infix proxy for Cons, and `[]` for Nil.

Let's suppose that, for update, if the key is unknown then the pair is added at the end of the list. Otherwise, the value associated with the key is modified:

```
(* a key/value update function *)
let rec update key value list =
 match list with
  [] → [(key,value)] |
  (key1,value1)::rest →
    if key=key1
    then (key,value)::rest
    else (key1,value1)::update key value rest;;
```

For example:

```
update "a" 1 [] ⇒...⇒ [("a",1)]
update "b" 2 [("a",1)] ⇒...⇒ [("a",1];("b",2)]
update "a" 3 [("a",1);("b",2)] ⇒...⇒ [("a",3);("b",2)]
```

Then, for search, if the key is found then the most recent associated value is returned. And if the key isn't found then some error condition arises:

```
let rec search key list =
 match list with
  [] → raise (Failure "search") |
  (key1,value1)::rest →
   if key=key1
   then value1
   else search key rest;;
```

We'll come back to the error condition later.

If memory space is not an issue, then, for update, rather than checking to see if the key is known, the list may be extended with a new tuple at the front, in a stack discipline:

```
let sUpdate key value rest = (key,value)::rest;;
```

For example:

```
sUpdate "a" 1 [] ⇒...⇒ [("a",1)]
sUpdate "b" 2 [("a",1)] ⇒...⇒ [("b",2];("a",1)]
sUpdate "a" 3 [("b",2);("a",1)] ⇒...⇒ [("a",3);("b",2);("a",1)]
```

Here, for search, the latest value associated with a key will always be found first. Update no longer requires searching and copying, but the list grows with the number of updates rather than the number of unique keys.

In contrast, for a given list of pairs, instead of using general update and search functions, we could construct a far more efficient list specific search function with the associations bolted in. For example, for:

[("a",1);("b",2)]

we could use:

```
let rec pSearch key =
 if key="a"
 then 1
 else
  if key="b"
  then 2
  else raise (Failure "search");;
```

This is as if we had unfolded the search function across the list.

```
let rec fun pSearch key = search key [("a",1),("b",2)] ⇒...⇒

let rec fun pSearch key =
 match [("a",1);("b",2)] with
  [] → raise (Failure "search") |
  (key1,value1)::rest →
   if key=key1
   then value1
   else search key rest ⇒...⇒

let rec fun pSearch key =
 if key="a"
 then 1
 else search key [("b",2)] ⇒...⇒
```

```
let rec fun pSearch key =
 if key="a"
 then 1
 else
  match [("b",2)] with
   [] → raise (Failure "search") |
   (key1,value1)::rest →
    if key=key1
    then value1
    else search key rest ⇒...⇒

let rec fun pSearch key =
 if key="a"
 then 1
 else
  if key="b"
  then 2
  else search key [] ⇒...⇒

let rec fun pSearch key =
 if key="a"
 then 1
 else
  if key="b"
  then 2
  else
   match [] with
    [] → raise (Failure "search") |
    (key1,value1)::rest →
     if key=key1
     then value1
     else search key rest ⇒...⇒

let rec fun pSearch key =
 if key="a"
 then 1
 else
  if key="b"
  then 2
  else raise (Failure "search")
```

This Functional Pearl explores how we might construct association pair specific functions like this, by effectively extending the function for a known sequence of pairs with a new layer of **if** ... **then** ... **else** ... for a new pair. To do so, we will build on the pioneering work of Christopher Strachey, who introduced such updateable functions for modelling states, stores and environments in denotational semantics. We shows that Strachey's functions can be used to implement a real world key/value update and search API, without recourse to data structures such as lists and trees.

2 Strachey's update function

In Strachey's approach (Strachey, 1966), the meanings of programs are characterised as functions from input to output domains, via intermediate domains representing various

forms of state. In turn, domains are modelled as updateable functions. Thus, Strachey characterised a store as a function from L-values to R-values:

$$\sigma' = U\alpha(\beta', \sigma)$$

where σ' is the new store function, U is the update function, α is an L-value, β' is an R-Value to be associated with α, and σ is the old store function.

Subsequently (Strachey, 2000), Strachey defined U as:

$$(U(\alpha, \beta'))\sigma = \sigma' \text{ where } \sigma'\chi = (\chi = \alpha) -> \beta', \sigma\chi$$

using the conditional expression notation:

$$condition -> expression_1 , expression_2$$

Recasting this in OCaml gives:

```
let rec strUpdate ident value store =
 fun id → if id=ident then value else store id;;
```

In effect, adding an identifier/value pair to a given store is realised by wrapping a conditional expression round the store function. Furthermore, just like our second list update function, the store function is extended in a stack discipline; when it is called with some argument identifier, the conditional for the most recent update for a known identifier is always met before those for prior updates.

For example, given an initial empty store function:

```
let empty id = raise (Failure "search");;
```

consider adding an association between "a" and 1:

```
strUpdate "a" 1 empty ⇒...⇒
fun id → if id="a" then 1 else empty id ⇒...⇒
fun id → if id="a" then 1 else raise (Failure "search")
```

Similarly, adding an association between "b" and 2 gives:

```
strUpdate "b" 2 (fun id → if id="a"
                          then 1
                          else Raise (Failure "search")) ⇒...⇒

fun id → if id="b"
         then 2
         else (fun id → if id="a"
                        then 1
                        else Raise (Failure "search")) id ⇒...⇒

fun id → if id="b"
        then 2
        else
         if id="a"
         then 1
         else Raise (Failure "search")
```

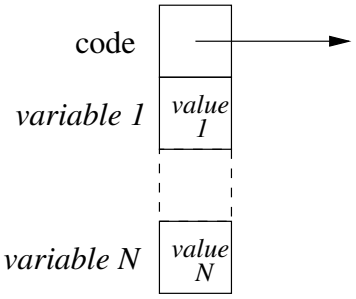

Fig. 1. Closure

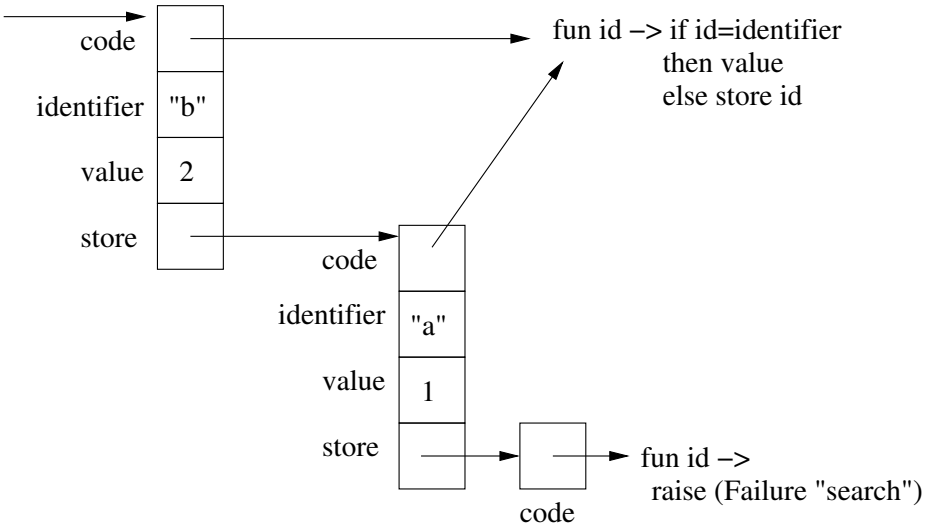

Fig. 2. Closure for `"a"/1, "b"/2`

3 Closures

We have reduced applications of the update function to something resembling a normal form, delving into the function body until we can perform no more substitutions. In functional language implementations, however, partial application typically involves a mechanism to associate bound variables with arguments such that, when bound variables are encountered in function bodies, the associated arguments are used. Thus, when a call returns a function in which former bound variables appear free, their bindings must be accessible when the function is subsequently used.

Typically, a structure called a closure is built to record the code for that function and its free variable bindings. We will represent closures as shown in Figure 1. For example, Figure 2 shows the closure for the function for the associations for ”a”/1 and ”b”/2. Note that different closures may share the same code.

We’re now going to explore how to build updateable function that follow other disciplines. Unlike Strachey’s update, however, our functions will enable both look-up and self-extension. In general, we will build functions that model associations between arbitrary keys and values. For a known key we will return the associated value, and for an unknown

key we will update the function with a new association. Hence, the functions must always be passed both a key and a value, and return both a value and a possibly updated function.

4 Queue function update

In our original list function, we added an new association at the end of the list of known associations. Thereafter, the list is searched from the start to the end, and if the required entry is not found, then it is added at the end. That is, the list is maintained in what we might term a queue discipline as it is extended at the back but searched from the front.

In our pure functional analogue, operationally, we want to splice in a new conditional expression for a new association pair at the end of the function for the known association pairs.

For example, given:

```
fun id → if id="a" then 1 else raise (Failure "search")
```

we would like the effect of:

```
qUpdate "b" 2 (fun id → if id="a"
                        then 1
                        else raise (Failure "search")) ⇒...⇒

fun id → if id="a"
         then 1
         else
          if id="b"
          then 2
          else raise (Failure "search")
```

We need to somehow tease the function apart and reconstruct it. However, we are not going to use reflection or meta-programming. Rather, the function being updated will reconstruct itself, association by association. Thus, at the end of the reconstructed chain of associations, rather than raising an exception, the function requires the ability to splice in a new association offering the possibility of further extensions in the future.

Suppose we have an update function:

```
fun rec qUpdate key value rest = ...
```

where `rest` is the function being updated.

For an unknown key, at the end of the chain of conditionals, there must be a call to a function that returns both the associated value and a new function that knows about the new association:

```
let rec qEmpty key value = (value,qUpdate key value qEmpty)
```

Note that `qEmpty` passes itself to `qUpdate`.

Now, `qUpdate` is to return a new look-up function for a new key and value. If the new function knows the key, it returns the value and recreates itself:

```
let rec qUpdate key value rest =
 fun k v → if k = key
           then (value,qUpdate key value rest)
           else ...
```

Otherwise, it:

- calls the old function to return the value for the unknown key and a possibly modified old function;
- returns the value and recreates itself using the possibly modified old function:

```
let rec qUpdate key value rest =
 fun k v → if k = key
           then (value,qUpdate key value rest)
           else
            let (newvalue,newrest) = rest k v
            in  (newvalue,update key value newrest)
```

Here the call to rest will return a copy of itself, possibly changed if the required key k is unknown.

For example, let's start by extending an empty sequence with the pair "a"/1:

```
qEmpty "a" 1 ⇒...⇒

(1,qUpdate "a" 1 qEmpty) ⇒...⇒

(1,fun k v → if k="a"
             then 1
             else
              let (newvalue,newrest) = qEmpty k v
              in (newvalue,qUpdate "a" 1 newrest))
```

Let's now add "b"/2:

```
(fun k v → if k="a" then 1 else ...) "b" 2 ⇒...⇒

let (newvalue,newrest) = qEmpty "b" 2
in (newvalue,qUpdate "a" 1 newrest) ⇒...⇒

let (newvalue,newrest) = (2,qUpdate "b" 2 qEmpty)
in (newvalue,qUpdate "a" 1 (qUpdate "b" 2 qEmpty) ⇒...⇒

(2,qUpdate "a" 1 (qUpdate "b" 2 qEmpty))
```

To recap, we started out with:

qUpdate "a" 1 qEmpty

with qEmpty as the rest function after "b"/2 and we now have:

qUpdate "a" 1 (qUpdate "b" 2 qEmpty)

with (qUpdate "b" 2 qEmpty) as the rest function after "a"/1.

In general, as a new pair is added, the function is reconstructed with the composed calls reflecting a queue order of access. The closure for "a"/1, "b"/2 is shown in Figure 3.

5 Types

The update function has a recursive type:

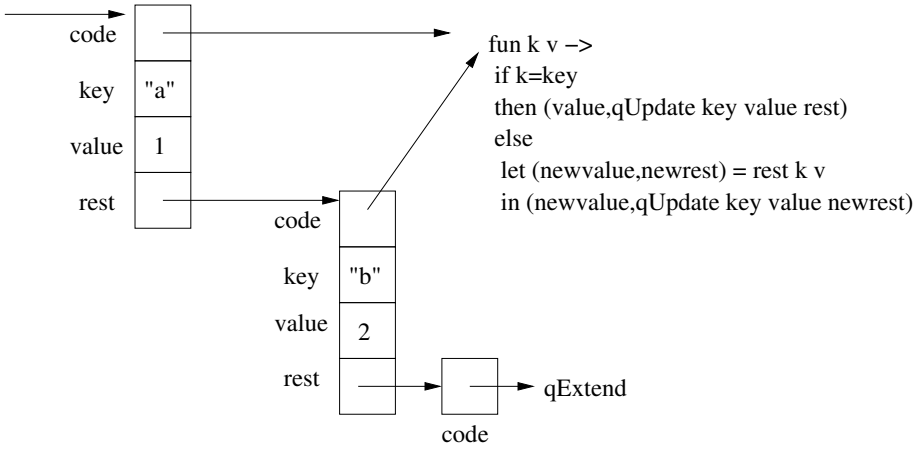

Fig. 3. Queue disciplined closure for `"a"/1, "b"/2`

$\alpha \rightarrow \beta \rightarrow U \rightarrow U$
where:
$U = \alpha \rightarrow \beta \rightarrow \beta * U$

assuming that α is an equality type.

Like the *Y* combinator, this update function cannot be implemented in a decidable, statically typed polymorphic language because any static type checker would infinitely recurse into type U. One approach for implementing Strachey functions is to use a dynamically typed language, where the decidability of the type correctness of U is not performed at compile time. For example, the empty and update functions in Python 2.7 are:

```
def qEmpty(key,value): return (value,qUpdate(qEmpty,key,value))

def qUpdate(rest,key,value):
    return lambda k,v: (value,qUpdate(rest,key,value))
                       if k==key
                       else qChange(rest(k,v),key,value)

def qChange((newvalue,newrest),key,value):
    return (newvalue,qUpdate(newrest,key,value))
```

A second approach is to use a language that permits type unsoundness by overlooking the cyclic property of type U, and this is the approach we take in this paper. We use the OCaml `-rectypes` flag to enable a selective occurs check admitting equirecursive types (Pierce, 2005). Thus, qEmpty and qUpdate have types:

```
val qEmpty : 'b → 'c → 'c * 'a as 'a = <fun>

val qUpdate : 'a → 'b
              → ('a → 'd → 'b * 'c as 'c)
              → ('a → 'd → 'b * 'e as 'e) = <fun>
```

Note that, for `qUpdate`, `'b` and `'d`, and `'c` and `'d` are the same types.

These functions can be implemented in System F compliant languages with a wrapper of user defined data type, but this is left as an exercise for the reader.

6 Ordered function update

We can modify the update function to give the effects of adding a new key/value association in key order. For a known key, we return the value and recreated function. For an unknown key, if it comes before the current known key we return the new value and recreate the function to reflect the new key order. Otherwise, as before, we seek the unknown key in the old function, and return the associated value and possibly modified function:

```
let rec oUpdate key value rest =
 fun k v → if k=key
           then (value,oUpdate key value rest)
           else
            if k<key
            then (v,oUpdate k v (oUpdate key value rest))
            else
             let (newvalue,newrest) = rest k v
             in (newvalue,oUpdate key value newrest);;

let rec oEmpty key value = (value,oUpdate key value oEmpty);;
```

Notice how, as each new pair is added, the function is reconstructed with the composed calls reflecting an ascending order of key access. See Figure 4 for successive closures for ”a”/1, ”c”/3, ”b”/2.

7 Ordered function update with key/value modification

Alternatively, we can modify the queue function to either search for the value associated with a key or change the value associated with a key:

```
let rec cUpdate key value rest =
 fun search k v →
           if k=key
           then
            if search
            then (value,cUpdate key value rest)
            else (v,cUpdate key v rest)
           else
            let (newvalue,newrest) = rest search k v
            in (newvalue,cUpdate key value newrest);;

let rec cEmpty search key value = (value,cUpdate key value cEmpty);;
```

Note that the size of this function only grows with the number of unique keys.

8 Unbalanced binary tree

We next construct an function that follows an unbalanced tree discipline, as recursive function calls rather than as a tree based data structure, with “rest” functions representing the left and right branches. If an unknown key comes before the current known key then it is sought using the left branch function; if it comes after that key then the right branch function is used. In either case, the whole function is recreated using the possibly changed left or right branch function as appropriate:

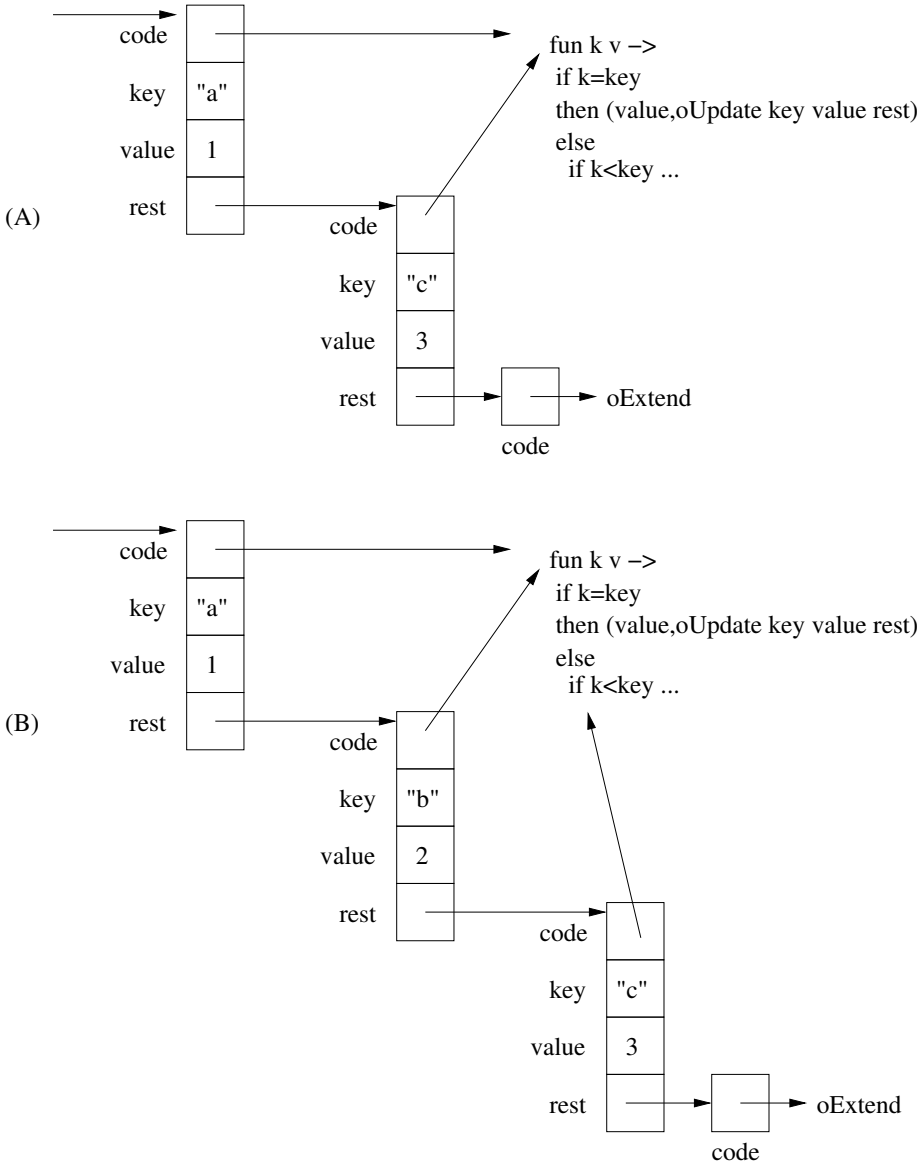

Fig. 4. Ordered closure for (A) `"a"/1, "c"/3` and (B) `"a"/1, "c"/3, "b"/2`

```
let rec tUpdate key value left right =
 fun k v → if k=key
           then (value,tUpdate key value left right)
           else
            if k<key
            then
             let (newvalue,newleft) = left k v
             in (newvalue,tUpdate key value newleft right)
            else
             let (newvalue,newright) = right k v
             in (newvalue,tUpdate key value left newright);;

let rec tEmpty key value = (value,tUpdate key value tEmpty tEmpty);;
```

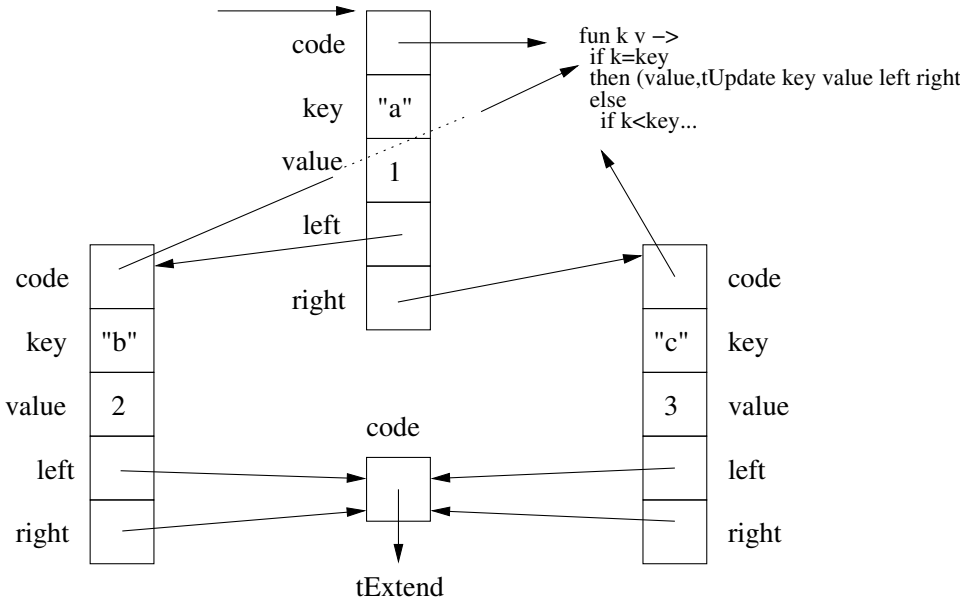

Fig. 5. Tree closure for `"b"/2, "a"/1, "c"/3`

The tree for ”b”/2, ”a”/1, ”c”/3 is shown in Figure 5.

9 Performance

We will now compare the performance of the list based and closure based functions for stack and queue disciplines. We are using the `camlopt` native code compiler within OCaml 3.11.2 running under Linux CentOS 6 on a 2.6GHz Intel Xeon with 22GB memory. Times are taken with the OCaml Sys.time function. Reported times are the averages of 5 runs.

Table 1 shows comparative times in seconds for first updating, and then searching the list and closure versions of the stack and queue disciplined functions with P unique pairs of strings and integers.

To make a fair comparison for the queue discipline, we introduce a new list queue function:

```
let rec lqUpdate k v l =
 match l with
 (key,value)::rest →
  if k=key
  then (value,l)
  else
   let (newvalue,newrest) = lqUpdate k v rest
   in (newvalue,(key,value)::newrest) |
 [] → (v,[(k,v)]);;
```

which has the same computation structure as the closure based function but uses list construction.

The list based update function (1) and list based search (2) both grow linearly with the number of pairs. This is because a new pair is always added at the end of the existing pairs.

	List based			**Strachey functions**		**Queue based**		
	update	search	update (stack)	update	search	update	search	update (list)
	`update`	`search`	`sUpdate`	`strUpdate`	`store`	`qUpdate`	`(qUpdate)`	`lqUpdate`
P	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
300	0.0018	0.0010	0.0000	0.0000	0.0004	0.0002	0.0028	0.0016
600	0.0068	0.0048	0.0000	0.0000	0.0010	0.0002	0.0140	0.0072
900	0.0166	0.0104	0.0000	0.0000	0.0022	0.0000	0.0380	0.0178
1200	0.0314	0.0184	0.0000	0.0000	0.0038	0.0000	0.0808	0.0336
1500	0.0546	0.0304	0.0000	0.0002	0.0062	0.0002	0.1406	0.0584
1800	0.0826	0.0434	0.0000	0.0002	0.0090	0.0002	0.2198	0.0880
2100	0.1164	0.0580	0.0002	0.0000	0.0122	0.0002	0.3297	0.1232
2400	0.1618	0.0774	0.0000	0.0002	0.0162	0.0004	0.4717	0.1796
2700	0.2154	0.0932	0.0000	0.0002	0.0194	0.0002	0.6379	0.2290
3000	0.2838	0.1208	0.0002	0.0000	0.0250	0.0004	0.8339	0.2846
3300	0.3583	0.1418	0.0002	0.0002	0.0296	0.0004	1.1342	0.3731
3600	0.4511	0.1730	0.0002	0.0002	0.0360	0.0006	1.5252	0.4739
3900	0.5497	0.2092	0.0000	0.0004	0.0426	0.0006	1.9151	0.5781
4200	0.6713	0.2298	0.0004	0.0002	0.0474	0.0008	2.2561	0.6971
4500	0.8019	0.2668	0.0004	0.0002	0.0562	0.0010	2.5478	0.8309
4800	0.9693	0.3006	0.0002	0.0004	0.0648	0.0010	2.9050	0.9998
5100	1.1296	0.3453	0.0006	0.0004	0.0714	0.0012	3.3735	1.1596
5400	1.3300	0.3975	0.0004	0.0004	0.0814	0.0010	3.9148	1.3544
5700	1.5160	0.4265	0.0004	0.0004	0.0910	0.0016	4.5797	1.5428
6000	1.7563	0.4859	0.0004	0.0006	0.1000	0.0014	5.2488	1.7811

Table 1. Times in seconds for list and closure based stack and queue functions

List based stack update (3) and the Strachey update function (4) both take negligible time. This is because the list function adds a cons cell at the top level where the Strachey function adds a closure at the top level.

Strachey's search function (5) takes around 20% of the time of list search (2). This is at first sight surprising as we might expect closure entry to be more costly than list selection. However, we suspect that OCaml is optimised to always place closure frames directly onto the stack with stack relative code in the closure body. The access time for both functions grows linearly with the number of pairs.

The queue based update function (6) takes a negligible time. Like the Strachey function, this function also adds a top level layer of closure to an existing function.

The queue based search function returned by `qUpdate` (7) takes around 10 times as long as list based search (2). Descending the chain of closures to find a required key is faster than descending the corresponding list, as shown by the comparison of the list (2) and Strachey (5) searches. However, returning the result from the end of the queue also involves reconstructing the entire chain of closures. Further investigation is needed to clarify the precise behaviour of the OCaml implementation here.

The list based queue update (8) is comparable to list based update (1).

10 Related work

Church introduced the representation of numerals (Church, 1936; Church, 1941) as pure λ functions (pp28-9). In general, the *n*th numeral is:

$\lambda a.\lambda b.a^n b$

that is it applies function *a n* times to some value *b*. Arbitrary numerals can be generated from zero and successor:

$zero = \lambda a.\lambda b.b$
$succ = \lambda a.\lambda b.\lambda c.b\ (a\ b\ c)$

so an application of *succ* adds a layer of applying *a*.

Church also introduced a representation of pairs (Church, 1941) with selectors as pure λ functions (p30):

$[M,N] = \lambda a.a\ M\ N$
$I = \lambda b.b$
$2_1 = \lambda a.a\ (\lambda b\ c.c\ I\ b)$
$2_2 = \lambda a.a\ (\lambda b\ c.b\ I\ c)$

Since Church, there have been many similar approaches to representing numerals as λ functions. In particular, that from Scott's widely cited but unpublished System of Functional Abstraction has been particularly influential. Curry et al (Curry *et al.*, 1972) seem to give the first account of Scott numerals, but using a mix of λ calculus and combinators (pp259-60):

$zero = A = K = \lambda x.\lambda y.x$
$succ = F = \lambda uxy.Bxyu$

Given $B = KI$, *succ* reduces to:

$\lambda uxy.KIxyu \Rightarrow \lambda uxy.Iyu \Rightarrow \lambda uxy.yu$

Parigot (Parigot, 1988), in a type theoretic exploration of recursive data types, first defines:

$zero = \lambda f.\lambda a.a$
$succ = \lambda v.\lambda f.\lambda a.((f\ v)\ (v\ f\ a))$

He also defines an alternative "stack" representation with:

$zero = \lambda f.\lambda a.a$
$succ = \lambda v.\lambda f.\lambda a.(f\ v)$

which is Scott's numeral.

Similarly, Steensgaard-Madsen (Steensgaard-Madsen, 1989) presents:

zero = λ*succ.*λ*zero.zero*
succ = λ*n.*λ*succ.*λ*zero.succ (n)*

citing Scott. He also presents a list representation.

Mogensen (Mogensen, 1994) builds on Scott's representation to provide an elegant if inefficient encoding of λ calculus in itself:

x = λ*abc.a x - identifier*
M N = λ*abc.M N - application*
λ*x.M* = λ*abc.c* (λ*x.M*) *- function*

so:

λ*x.x x* = λ*abc.c* (λ*x.*(λ*abc.b* (λ*abc.a x*) (λ*abc.a x*)))

Koopman et al (Koopman *et al.*, 1994) use Clean to explore the computational efficiency of Church, Scott and Parigot style representations of Peano numbers, lists and trees. They suggest that such encodings are a poor basis for practical implementations, and the Church style encoding is markedly less efficient than the simpler Scott and Parigot schemes.

These representations derived from the Church encoding share, with Strachey's and our approach, the use of closures to represent structures. They also share the feature of being composed from constructors with abstractions to enable the subsequent insertion of other functions to extend them or select from them. Thus, they treat structures as passive entities, to be manipulated by other functions, much as data structures are treated in programming languages. In contrast, our functions are active entities which, in combining construction and inspection, return modified copies of themselves.

(Michaelson, 1988) discusses the representation of data structures as closures using the Navel derivative of strict, ad-hoc polymorphic SASL (Turner, 1976). (Michaelson, 1994) analyses the constant space update function in the context of the direct functional implementation of the denotational semantics of a simple imperative language.

11 Conclusion

We have shown how to construct updateable elementary linear and branching data structures from closures using a minimal pure subset of the functional language OCaml. We have been inspired by Strachey's update function for representing maps in denotational semantics.

Search using the function from our queue update function is markedly slower than the equivalent list based search function. This suggests that search based on our other update functions will also be slower as they follow the same pattern of closure reconstruction in search.

However, it is pleasing that Strachey's update returns a function which is faster than the equivalent list function: a tribute to OCaml as well as to Strachey.

It would be interesting to explore how to construct updateable circular data structures using the same approach.

Acknowledgements

This work started at the CSIRO Division of Information Technology and the University of Technology, Sydney.

We're pleased to acknowledge the support of EPSRC EP/K009931/1 "Rathlin". The OCaml implementation of all functions in this paper is available online [1].

We would like to thank:

- Jamie Gabbay and Joe Wells for discussion about the types of these functions;
- Bob Tennant for discussion about the origin of Strachey's update function;
- the anonymous referees for constructively critical comments.

References

Adams, S. (1993). Efficient sets - a balancing act. *Journal of Functional Programming*, **3**(4), 553–561.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, **58**, 345–363.

Church, A. (1941). *The calulii of λ conversion*. Princeton University Press.

Curry, H. B., Hindley, J. R., & Seldin, J. P. (1972). *Combinatory Logic: Volume II*. North-Holland.

Koopman, P., Plasmeijer, R., & Jansen, J. M. (1994). Church Encoding of Data Types Considered Harmful for Implementation: Functional Pearl. Tobin-Hochstadt, S. (ed), *Proceedings of 26th International Symposium on Implementation and Application of Functional Languages*. ACM.

Michaelson, G. (1988). *Data structures as self-modifying functions*. Tech. rept. TR-FB-88-09. Division of Information Technology, CSIRO, North Ryde, Australia.

Michaelson, G. (1994). Association sequences as self-modifying functions. *Journal of Programming Languages*, **2**(1), 41–66.

Mogensen, T. Æ. (1994). Efficient Self-Interpretaion in Lambda Calculus. *Journal of Functional Programming*, **2**, 345–364.

Parigot, M. (1988). Programming with proofs: a second order type theory. *Pages 145–159 of:* Ganzinger, H. (ed), *Proceedings of ESOP'88: 2nd European Symposium on Programming*.

Pierce, B. (2005). *Advanced Topics in Types and Programming Languages*. MIT Press.

Steensgaard-Madsen, J. (1989). Typed representation of objects by functions. *ACM Transactions on Programming Languages and Systems*, **11**(1), 67–89.

Strachey, C. (1966). Towards a Formal Semantics. *Pages 198–220 of:* Steele, T. B. (ed), *Formal Language Description Languages For Computer Programming*. North-Holland.

Strachey, C. (2000). Fundamental Concepts in Programming Languages. *Higher Order and Symbolic Computing*, **13**, 11–49.

Turner, D. (1976). *SASL Language Manual*. Tech. rept. University of St Andrews.

[1] `https://github.com/robstewart57/strachey-ocaml`