Monadic Reference Counting

Giuseppe Maggiore Michele Bugliesi
Università Ca' Foscari Venezia
Dipartimento di Informatica
{maggiore,bugliesi}@dsi.unive.it

Abstract

In this paper we show how a powerful memory and resource management technique such as reference counting can be implemented transparently as a library through the use of monads. While this is interesting in itself, since garbage collectors are not always the best choice for all programs, our paper also shows how the bind and return operators can be used to correctly identify the lifetime of resources and references. Finally, we discuss how with a powerful enough type system and the use of a parameterized monad we can track various interesting properties about the state of stateful computations. In our case we track the set of resource-types that the program handles, but we argue that in other cases this technique could be used to track even more complex and interesting properties.

1 Introduction

Modern computer languages are very reliable when it comes to writing a large class of common, real-world applications. For example, relatively simple form applications or web sites can be built extremely easily in languages such as Java, C# and many others. This is thanks to commonplace facilities like garbage collectors, classes and inheritance and large libraries which simplify many tasks which otherwise would be hard or error-prone. On the other hand, there is a not so small set of applications for which these languages do not perform even nearly as well; for example games, even though very powerful libraries such as XNA make them easier to write by encapsulating many useful patterns, are not so suitable for modern languages. For this reason most games are still written in C++ (sometimes even in C) and the transition to higher level languages is not happening as fast as it could. As another example we could consider mobile applications. The widespread adoption of very powerful, fully programmable smartphone like the iPhone, Google Android or Windows Phone 7 makes performance even more important to achieve: lighter applications mean much better applications where CPU cycles and battery are both scarce resources. On the other hand, to allow as many developers as possible to easily create applications for these platforms, it makes sense (as indeed it is happening) to allow programming these devices with as languages that are as high-level as possible. Finally, there are many real time or soft real time applications that migh benefit from using high level languages but which cannot afford the pauses or slowdowns that sometimes the garbage collector might require, especially on less powerful hardware.

In this paper we document the results of implementing a videogame in such a high-level language (F#) for a mobile device. Early in our development cycle we discovered that the biggest problem of our application was that by continually allocating and deallocating instances of the same types (such as projectiles and particles) we kept triggering the garbage collector, forcing the application to a crawl everytime it started. Rather than implement object pooling, we decided to try and generalize this technique by implementing reference counting [9] inside the state monad [45]. The state monad contains and manages all the actual instances of the objects which require reference counting, and references to these objects can only be accessed through the state monad itself. This allowed us to track the lifetime of our entities, in a lightweight enough fashion to ensure the increase in performance that we needed and transparently enough that it became convenient to track other resource types as well, such as files or GPU memory which require manual disposal.

In this paper we discuss a possible generalization of the work described above, where in addition to a monadic reference counting system we also discuss how type-level meta-programming [32, 6, 26, 39] and the parameterized monad [1] can be used to track the types of resources with a strongly typed heterogeneous list [29], thereby removing even the need for the developer to "discover" what type the state must have. Also, this system could be the stepping stone for a more refined state-tracking monad which uses type-level values and phantom types to track expected properties of the state monad at various points in the program.

An important note is about the computer language in which we have written the samples. We have used a pseudo-Haskell, and we believe our listings may be turned into a working program without too much effort. This notwithstanding, it must be noted that said effort has not been made by us: our software system is mostly interested in performance, and from the point of view of benchmarking performance Haskell may not be the best suited language given its lazy evaluation strategy. For us this pseudo-Haskell has acted as a clear and easily implementable specification, and we do not claim anything about its workability in any present or future versions of the language.

2 State Monad

The state monad [45, 28, 43, 37, 46] is a monad which allows us to write stateful, imperative computations in a pure language without sacrificing purity. A value of state monad type represents a statement, and its type requires that a value of type *state* is passed as a parameter to the statement in order for it to evaluate to its result. Evaluating a statement not only returns the result of the evaluation, but also the new value of the state: the input state may have been modified, that is the evaluation of any statement may produce side-effects.

The type of the state monad reminds the type of the denotational semantics of an imperative statement. This is interesting, in that we could consider values of the state monad as the denotation of statements:

```
type St s a = s -> (a,s)
```

We can bind statements together into composite statements and return values inside statements. When binding, we concatenate the two statements by evaluating the first and then plugging its result into the second and then evaluating it:

```
(>>=) :: St s a -> (a -> St s b) -> St s b
p >>= k = \s ->
  let y,s' = p s
  in k y s'
```

When we wish to pack a value x inside a monad we return it:

```
return :: a -> St s a
return x = \s -> x,s
```

3 Reference

The state monad as presented above is well-known and commonly used in pure languages such as Haskell. This kind of world- passing-style (or store-passing-style) is powerful and allows a programmer to write perfectly fine imperative code. Since this monad is mostly (if not always exclusively) used to pass around the state and manipulate mutable portions of the state through the use of references and proxies, we now propose an extension of the monad that focuses exclusively on these references. A system resource is anything that we wish to release as soon as we are done with it. System resources may include streams, network connections, GPU memory in GPGPU applications (as done, for example, in [34, 10]), threads, etc. In some cases even memory references may be treated as resources, especially when we wish to recycle memory as soon as possible rather than just leave the job to the garbage collector. A system resource can be defined in terms of an appropriate type class:

```
class Reference f a s where
  new :: a -> St s (f a)
  incr :: f a -> St s ()
  decr :: f a -> St s ()
  get :: f a -> St s a
  count :: f a -> St s Int
```

A Reference is represented by a functor f and a type a. a is the type of our actual resource, and f a is the proxy that we will use to access this resource. The proxy f a may:

- be created from a value a with new
- increment its internal counter with incr
- decrement its internal counter with decr
- get its internal value with get
- get its internal counter with count

The only public functions that we leave accessible are new and get. The other functions can only be called from inside the monad implementation.

3.1 Reference axioms

A Reference has some requirements that it must respect. These requirements are expressed in terms of axioms, which are a way to formalize the most obvious expectations we have towards a Reference. Informally, we require that a Reference:

- starts with a count of 1 after being created with new

- returns a value with get when its count is greater than 0; otherwise get returns ↑
- incr and decr respectively increment and decrement the internal count by 1, but only if the count is greater than 0; otherwise they both return ?

We can express these requirements with the help of the do-notation as:

```
forall s x . (do r <- new x
                 count r) s = (1,_)

forall s r . (get r) s = (_,_) if (count r s) = (c,_) with c > 0
                         ↑ otherwise

forall s r . (do incr r
             count r) s = (c+1,_) if (count r s) = (c,_) with c > 0
                         ↑ otherwise

forall s r . (do decr r
                 count r) s = (c-1,_) if (count r s) = (c,_) with c > 0
                              ↑ otherwise
```

We also expect, but this is really up to the implementation, that whenever the internal count of a resource is decremented to 0, then the resource will be freed.

3.2 Possible reference implementation

Our definition of Reference may appear a bit excessive. What is the meaning of the functor f? f is the type of references, which will be used as proxies of the actual values of type a. The idea behind the Reference type class is that we do not force anything about how references are represented: the type of references depends strongly on the type of the heap inside which the reference will point to. To strengthen this intuition, let us discuss a few possible implementations. A heap may be a list of values where we add values to the end of the list; references are the index from the beginning of the list to the appropriate item and its associated counter:

```
type F a = Int
instance Reference F a [(Int,a)]
  Ě
```

A heap may be a list of lists of values, for performance reasons (indexing twice helps skipping many elements):

```
type F a = (Int,Int)
instance Reference F a [[(Int,a)]]
  ...
```

A heap may even be a more elaborate data structure, such as a map from a key k into a value a and its associated counter:

```
instance (Map s, k ~ Key s) => Reference k a (Map k (Int,a))
  ...
```

where Map s means that s is a map and Key s is a type function that returns the type of key used to index elements of s; is the binding operator for type variables.

The above definitions all define mappings from keys to values that are all pure and quite obvious. Since we are not limiting our language to a pure functional one (the thing will have to run on imperative hardware after all) it is not at all inadmissible that the implementation of the heap and its reference may somehow rely on pointers and mutable state. A simple yet effective implementation only requires that our language supports arrays. References are indices in the array, but this time the access will be much faster:

```
type F a = Int
instance Reference F a [|(Int,a)|]
  ...
```

or

```
type F a = Int
instance Reference F a [|[|(Int,a)|]|]
  Ě
```

where [| a |] is an array of a. We could optimize this last definition further (it has very high performance and is very easy to implement, and as such we have used it in our benchmarks) by adding to each element of the external array the number of free items (those with the counter set to 0):

```
type F a = Int
instance Reference F a [|(Int,[|(Int,a)|])|]
  ...
```

Of course whatever implementation we pick for managing references, we must keep in mind that incrementing (decrementing) a reference to a value also requires us to inductively increment (decrement) all the references contained inside that value. For this reason we modify the Reference class so that its incrementation and decrementation functions only do one step of incrementing and decrementing, while the recursive work is left to a new pair of incr and decr functions:

```
class Reference f a s where
  new :: a -> St s (f a)
  incrStep :: f a -> St s ()
  decrStep :: f a -> St s ()
  get :: f a -> St s a
  count :: f a -> St s Int
```

We define, in the spirit of the LIGD library ([REFERENCE]), a representation datatype using GADTs (Generalized Algebraic Datatypes):

```
data Unit = Unit
data Sum a b = Inl a | Inr b
data Prod a b = Prod a b

data Rep t where
  RUnit :: Rep Unit
  RSum :: Rep a -> Rep b -> Rep (Sum a b)
  RProd :: Rep a -> Rep b -> Rep (Prod a b)
  RType :: Rep c -> EP b c -> Rep b
```

where the EP datatype is the witness of the isomorphism between two type b and c and is defined as:

```
data EP b c = EP { from :: (b -> c), to :: (c -> b) }
```

As an example, let us see how we could define a generic equality function:

```
geq :: Rep a -> a -> a -> Bool
geq (RUnit)       Unit          Unit          = True
geq (RSum ra rb ) (Inl a1 )     (Inl a2 )     = geq ra a1 a2
geq (RSum ra rb ) (Inr b1 )     (Inr b2 )     = geq rb b1 b2
geq (RSum ra rb ) _             _             = False
geq (RProd ra rb) (Prod a1 b1 ) (Prod a2 b2 ) = geq ra a1 a2 && geq rb b1 b2
geq (RType ra ep) t1            t2            = geq ra (from ep t1) (from ep t2)
```

The generic equality function takes an additional parameter which is the representation of the type of the values being compared. This allows us to index the function based on the type (in fact we say that geq is a TIF, or "type-indexed-function").

Now, consider how we could build the representation of a list. We start with the embedding projection:

```
fromList :: [a] -> Sum Unit (Prod a [a])
fromList [] = Inl Unit
fromList (a:as) = Inr (Prod a as)
toList :: Sum Unit (Prod a as) -> [a]
toList (Inl Unit) = []
toList (Inr (Prod a as)) = a:as
```

then we write the representation using RType:

```
rList :: Rep a -> Rep [a]
rList ra = RType (RSum RUnit (RProd ra (rList ra)))
            (EP fromList toList)
```

At this point we define two intermediate functions incrRep and decrRep that recursively invoke themselves in order to invoke incrStep and decrStep respectively on each Reference found inside the original Reference:

```
incrRep :: Reference f a s => Rep a -> (f a) -> St s ()
incrRep (RUnit) ref = incrStep ref
incrRep (RSum ra rb) ref  =
  do v <- get ref
     case v of
     | Inl x -> incrRep ra x
     | Inr x -> incrRep rb x
     incrStep ref
incrRep (RProd ra rb) ref =
  do (x,y) <- get ref
     incrRep ra x
     incrRep rb x
     incrStep ref
incrRep (RType ra ep) ref =
  do v <- get ref
     incrRep ra (from ep v)
     incrStep ref

decrRep :: Reference f a s => Rep a -> (f a) -> St s ()
...
```

We omit the body of decrRep since it is substantially identical to that of incrRep, to the point that both functions could be easily defined in terms of a single combinatory (a monadic version of the everywhere function [REFERENCE]).

We instance a "constant" Reference so that a simple value can be interpreted as a Reference :

```
type Id a = a
instance Reference Id a s where
  new = return
  incrStep = return ()
  decrStep = return ()
  get = id
  count = 1
```

At this point we define a typeclass that captures all the datatypes representable in terms of the above GADT:

```
class Representable a where
  rep :: Rep a
```

We also require that a Reference is always to a Representable datatype:

```
class Representable a => Reference f a s where
  ...
```

in order that the representation is implicit in the Reference and must not be passed around each time. At this point we can define the actual incr and decr functions:

```
incr :: Reference f a s => f a -> St s ()
incr ref = incrRep rep ref

decr :: Reference f a s => f a -> St s ()
decr ref = decrRep rep ref
```

4 Bind, return and lifetime

Let us now focus on the notion of variable scope that is implicit in a monad. Whenever we bind two statements, the scope of the bound value is limited to the body of the second parameter (unless it is returned). This means that after the bound value is passed to the second parameter, then its lifetime is exhausted and the value may be decremented. Of course, whenever we return a value then to prevent its premature reclamation we will increment it to counter its decrementing by the enclosing binding. The new type of the bind and return operators now requires that these two only manipulate monads to references. Bind will also decrement the bound value as soon as it goes out of scope:

```
(>>=) :: Reference f a s => St s (f a) -> (f a -> St s b) -> St s b
p >>= k = \s ->
  let y,s' = p s
  let z,s'' = k y s'
  in z,snd(decr y s'')
```

while return will increment its parameter:

```
return :: Reference f a s => f a -> St s (f a)
return x = \s -> x,snd(incr x s)
```

This is convenient because in the rest of the paper we will have no need to use those as standalone monadic statements and instead we will use them only as functions from state to state.

4.1 General recursion

Whenever we are in the presence of stateful recursive functions, then we may find that some embarrassing facts occur. In particular, the lifetime of a local value inside the body of the recursive function may be unnaturally lengthened to encompass the entire sub-trees of the recursive call. This is unacceptable, especially if we think about functions that recursively open a lot of files (like when traversing the file system in search for something) or use a lot of memory.

4.1.1 Benchmark

Let us consider a very challenging example of this scenario. We wish to create a balanced binary tree from a set of points:

```
bt :: [Point] -> Tree [Point]
bt pts =
  if size pts < 1000 then mk_leaf pts
  else
    let m = median pts
    let l,r = split pts m
    let tl = bt l
    let tr = bt r
    in mk_node (tl,tr)
```

In this example we can clearly see that until both calls to bt l and bt r are completed, then we may not release any memory at all! This is clearly nonsense, since pts can be released right after the call to split and l can be released right after the first recursive call to bt.

4.1.2 Explicit continuations

Enter Continuation Returning Style. We try and solve this problem with trampolines [3], that is intermediate pieces of code that are "wrapped" around our recursive calls. We will call this style Continuation Returning Style because statements in this style do not return their result when executed but rather return another statement which, when executed in its turn, will complete the job. We refer to this nested statement as a trampoline. We define trampolines as statements that capture by (explicit) closure a containing Reference which gets incremented when the trampoline is created and which is released after the trampoline is executed. Using trampolines does not exclude the possibility of using the state monad as defined above. Whenever its conservative notion of lifetime is acceptable, we will be free to use it; whenever its notion of lifetime is too restrictive, then we will use our trampolines and jump between the two easily. A trampoline is defined as:

```
type Trmp s a = St s (St s a)
```

It may help understanding trampolines in terms of the following diagram which shows the order in which the state flows:

A trampoline is constructed from the captured values that will have a lifetime at least as long as that of the trampoline and the actual body of the trampoline. The first thing the trampoline constructor does is increment the captured values, and then it binds the execution of the body of the trampoline with decrementing the captured values:

```
trmp :: Reference f a s => f a -> (f a -> Trmp s b) -> Trmp s b
trmp ctxt p = \s -> (\s ->
  let p',s' = p ctxt s
  in p' (snd (decr ctxt s'))), snd (incr ctxt s)
```

We have a return function that creates a trampoline. To make this work we assume that our language supports operator overloading, where the most specific overload will be invoked at each binding site:

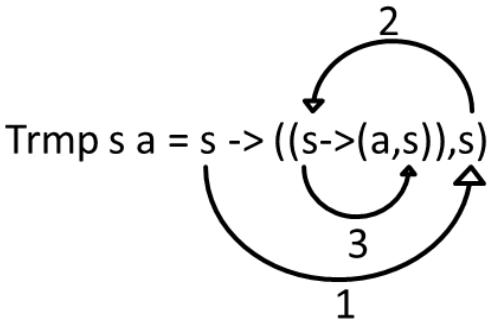

Figure 1: State flow

```
return :: Reference f a s => f a -> Trmp s (f a)
return x = \s->(\s -> x,snd (incr x s)),s
```

We can turn a trampoline into a statement with relative ease by unpacking it and executing it twice:

```
(!) :: Trmp s a -> St s a
!p = \s ->
  let p',s' = p s
  in p' s'
```

Even more interesting is the fact that we can easily bind values of the state monad with trampolines; the result will be another trampoline. This kind of binding starts decrementing the bound value y much sooner than the standard binding, since y get decremented before the full execution of k. The idea is that k has a chance to increment y as its ctxt, and right after this has been done k y returns k'. The actual execution of k is thus stored as k', which may contain recursive calls to some caller. Before executing this (possibly time-consuming) code we get our chance to free y in case it were not needed anymore inside k':

```
(>>=) :: Reference f a s => St s (f a) -> (f a -> Trmp s b) -> Trmp s b
p >>= k = \s ->
  let y,s' = p s
  let k',s'' = k y s'
  in k', decr y s''
```

Notice that this version of the binding operator is exactly the same as the original binding operator, and since trampolines are state monads then there is no need for the explicit definition.

4.1.3 Solving the benchmark problem

We can rewrite the binary tree example above as:

```
bt :: Reference f [Point] s => f [Point] -> Trmp s Tree [Point]
bt pts =
  if size pts < 1000 then trmp pts (\pts -> return mk_leaf pts)
  else
    trmp pts (\pts ->
      do m <- median pts
         trmp (pts,m) (\(pts,m) ->
           do l,r <- split pts m
              trmp (l,r) (\(l,r) ->
              do tl <- bt l
                 trmp (tl,r) (\(tl,r) ->
                 do tr <- bt r
                    trmp (tl,tr) (\(tl,tr) ->
                      return mk_node (tl,tr))))))
```

where we can clearly see that each continuation declares explicitly what it will keep alive (in terms of reference counting). This is a similar style to the well-known world-passing-style of the state monad, but we are also passing around the active scope.

4.1.4 Syntactic sugar for continuations

It is very important to notice a detail that threatens the correctness of our system. Continuations may not capture values whose type respects the Reference predicate; continuations may only use those counters that they explicitly captured, otherwise we have no guarantee that the captured counter will still be valid when accessed. For this reason we introduce a notion of syntactic sugar for expressing our continuations where the captured variables are the free variables of type Reference accessed in the body of the continuation. We tell the compiler to search for these variables with the trmp keyword (not to be confused with the private trmp function seen above). The translation rule is quite straightforward:

```
        [| trmp x <- m n |] = trmp ctxt (\ctxt -> m >>= fun x -> [| trmp n|])
        where ctxt = FC(m) ∪ FC(n)

        [| trmp return m |] = trmp ctxt (\ctxt -> return m)
        where ctxt = FC(m)
where FC(t) = {x:f a ∈ FV(t) : ∃s . Reference f a s}
```

The resulting code is:

```
bt :: Reference f [Point] s => f [Point] -> Trmp s Tree [Point]
bt pts =
  if size pts < 1000 then trmp return mk_leaf pts
  else
    trmp m <- median pts        -- FC = pts
       l,r <- split pts m       -- FC = m,pts
       tl <- bt l               -- FC = l,r
       tr <- bt r               -- FC = r,tl
       return mk_node (tl,tr)   -- FC = tl,tr
```

and since we can easily see that:

```
FC(trmp m <- median pts
        l,r <- split pts m
        tl <- bt l
        tr <- bt r
        return mk_node (tl,tr)) = pts
FC(trmp l,r <- split pts m
        tl <- bt l
        tr <- bt r
        return mk_node (tl,tr)) = (pts,m)
FC(trmp tl <- bt l
        tr <- bt r
        return mk_node (tl,tr)) = (l,r)
FC(trmp tr <- bt r
        return mk_node (tl,tr)) = (tl,r)
FC(return mk_node (tl,tr)) = (tl,tr)
```

then it is clear how the sample with implicit continuations becomes identical to the one with explicit continuations.

5 Parametrized state monad

We now move to a more powerful definition of the state monad, the parametrized state monad [1]. This new version of the monad allows statements to make (static) changes to the state type, rather than just (dynamic) changes to the state value. The parametrized state monad has the following type:

```
type St p q a = p -> (a,q)
```

The definition of a counter can now take advantage of the knowledge that all the proxies f a point to the same type of storage (lists, lists of lists, maps, arrays, etc. as seen in Reference implementation), and a value with the storage type must be present in the state whenever we manipulate a proxy:

```
class Reference f a where
  Storage f a :: *
  emptyStorage :: Storage f a
  new :: (s_a ~ Storage f a, Addable s_a s, s_a ∈ s_a .+ s) => a -> St s (s_a .+ s) (f a)
  incr :: (s_a ~ Storage f a, s_a ∈ s) => f a -> s -> s
```

```
  decr :: (s_a ~ Storage f a, s_a ∈ s) => f a -> s -> s
  get :: (s_a ~ Storage f a, s_a ∈ s) => f a -> St s s a
  count :: (s_a ~ Storage f a, s_a ∈ s) => f a -> St s s Int
```

In particular Storage f a is the type function that associates the type of proxies with the type of actual containers. We also define the value of the empty storage with the property emptyStorage. The new function requires that the appropriate storage gets added to the input type of the state; the (idempotent) addition of an element to a heterogeneous list [29, 23] is the .+ operator. The added item must be available in the resulting type, and to ensure this we use the ∈ predicate. incr and decr both require that the storage is available in the manipulated state (otherwise no incrementing and decrementing could happen because there would be no "slot" to perform the computation in). Similarly we define get and count. We also define a convenient type-class for types with a default value; this way we can define the default value of a storage as its emptyStorage:

```
class Default x where
  default :: x

instance Reference f a => Default (Storage f a) where
  default = emptyStorage
```

Now we can fill the gaps of the new definition of the Counter type class. A type x can be added to a type s (the result is s .+ x) if it respects the Addable predicate:

```
class Addable x s where
  s .+ x :: *
  add :: s -> s .+ x
```

A type x is part of the heterogeneous list s if it respects the ∈ predicate; this predicate has two instances with respect to the heterogeneous lists constructor (.:):

```
class x ∈ s where
  lift :: (x -> x) -> (s -> s)

instance x ∈ x .: s
  lift f = \(x .: s) -> (f x) .: s
instance x ∈ s => x ∈ y .: s
  lift f = \(y .: s) -> y .: (lift f s)
```

When we wish to add a type x to another type s, we need to check if x∈s; if this is the case, then the addition is simply the identity with respect to s (x is already in s). If x?s then the addition returns a heterogeneous list with x as the head and s as the tail, and the value of the head is the default value of x:

```
instance (Default x, x ∈ s) => Addable x s where
  s .+ x = s
  add = id

instance (Default x) => Addable x s where
  s .+ x = x .: s
  add s = default .: s
```

At this point we can define the "regular" binding operator. When binding we need to be able to decrement the bound value of type f a inside the final state, which in our case has type r. For this reason we require that Storage f a∈r, so that we will be able to lift the decrementing operation from Storage f a->Storage f a into r->r:

```
(>>=) :: (Reference f a, Storage f a ∈ r) => St p q (f a) -> (f a -> St q r b) -> St p r b
s >>= k = \p ->
  let x,q = s p
  let y,r = k x q
  in y,lift (decr x) r
```

We omit the adaptation of trampolines to the parametrized monad as it is relatively straightforward.

As a side note, it is worth realizing that a big part of the above code, especially the ∈ predicate, will not work in current incarnations of the Haskell language and will rather give problems that can be solved (albeit in a rather verbose fashion) as seen in the [29]. Seen that we have used the above definitions to implement our own custom meta-programming library in F#, this has not been a problem for us.

6 Related work

6.0.5 Region-based memory management

Tofte and Talpin [41] present an inference system for classifying all allocated data of a program into regions and deducing a safe lifetime for each region, which enables provably memory-safe implementations of ML-like languages with-out a garbage collector. Crary et al.'s Capability Calculus [12] extends this work by allowing explicit region allocation and deletes, while making sure that all data accesses to a region happen during its lifetime. The commonality of these systems is that only regions are treated linearly; all other objects are allocated within regions and have types akin to guarded types. Regions are not first-class values and cannot be stored in data structures.

6.0.6 Linear type systems

Starting with Wadler [42], linear types systems have been used in purely functional languages to enforce single threading on the state of the world or to implement operations like array updating without the cost of a full copy. Linear type systems enable resource management at the granularity of a single object. Every use of an object of linear type consumes the object, leading to a programming style where linear objects are threaded through the computation. Wadler's let! construct, or its variations, can be used to give a temporary nonlinear type to an object of linear type. Walker and Watkins [47] study a type system with three kinds of objects: linear, reference counted, and region allocated. The kind of an object is fixed at allocation without a means to change kind. They provide let! only for regions.

6.0.7 Lighweight static capabilities

Static capabilities have been implemented by Kiselyov et al. [30] in a lightweight fashion in modern functional languages such as OcaML and Haskell. They propose a "style" of programming with three ingredients:

- A compact kernel of trust that is specific to the problem domain.
- Unique names (capabilities) that confer rights and certify properties, so as to extend the trust from the kernel to the rest of the application.
- Static (type) proxies for dynamic values.

The requirements imposed on the host language to implement this style are an expressive core language, higher-rank polymorphism and phantom types. Capabilities are represented as types; safety conditions are stored in types as in dependent-type programming. If a program type-checks, then the type system and the kernel of trust together verify that the safety conditions hold in any run of the program. In most cases, this static assurance costs us no run-time overhead.

6.0.8 Lightweight Monadic Regions

Kiselyov et al. [33] also build a library that statically ensures the safe use of resources such as file handles. They statically prevent accessing an already closed handle or forgetting to close it. The libraries can be trivially extended to other resources such as database connections and graphic contexts. Their library supports region polymorphism and implicit region subtyping, along with higher-order functions, mutable state, recursion, and run-time exceptions. A program may allocate arbitrarily many resources and dispose of them in any order, not necessarily LIFO. These monadic regions are implemented in Haskell as monad transformers. For contrast, the authors also implement a Haskell library for manual resource management, where deallocation is explicit and safety is assured by a form of linear types. The linear typing is implemented in Haskell with the help of phantom types and a parameterized monad to statically track the type-state of resources.

6.0.9 Strongly Typed Memory Areas

Jones et al. [13] discuss how to make Haskell suitable for systems programming tasks -including device driver and operating system construction. As a result of some gaps in functionality it often becomes necessary either to code some non-trivial components in more traditional but unsafe languages like C or assembler, or else to adopt aspects of the foreign function interface that compromise on strong typing and type safety. Some of these gaps may be filled by extending a Haskell-like language with facilities for working directly with low-level, memory-based data structures. The authors designed and implemented language features that allow programmers to de?ne strongly typed, high-level views, comparable to programming with algebraic datatypes, on the underlying bitdata structures. A critical detail in making this work is the ability to specify bitlevel layout and representation information precisely and explicitly; this is important because the encodings and representations that are used for bitdata are often determined by third-party speci?cations and standards that must be carefully followed by application programmers and language implementations.

7 Conclusions and future work

Modern computer languages are extremely powerful and their benefits to creating correct programs are widely accepted. Many tools used in these languages (garbage collection as the most prominent example) solve many problems and affords greater expressivity, but have a cost in terms of performance and cannot capture some interesting and useful patterns. Monads (the parametrized state monad in particular) can be a great tool for adding powerful capabilities such as memory pooling, reference counting for timely resource disposal and even additional forms of static analysis like ownership of shared variables in multiple threads, initializing variables before using them, and so on. While monads can indeed express some of these constructs very efficiently, others require further work. When building our system (in F#) we were forced to make use of quotations to process our code before execution, thereby adding a layer of program transformation to automate certain operations that depend on the structure of the type of their parameters; for this reason we believe that the presence of Haskell-style type classes would be the ideal complement to monads for creating libraries which complexity until now has forced them to reside in the runtime of the language (the virtual machine, for example) forcing all developers to use exactly one implementation. Having a system expressive enough so that the memory management strategy can be changed from one program to the next or can even be abstracted away could lead to very interesting consequences in terms of developer freedom when using high-level languages: does one need fast and automated memory management for non-cyclic data structures? Use a reference counting library. Does one need a traditional garbage collector? Use a garbage collection library. Does one need manual memory management? Use a manual memory management library. And so on. It might also be extremely powerful to be able to write programs where various styles of resource management are mixed, according to the needs and expected execution frequencies of the various portions of code.

We believe that our system, which is in its very early stages, could be greatly extended. One obvious direction of further work is to support more forms of static analysis inside the parameterized monad, from abstract interpretation to simple state machines that govern the behavior of certain entities up to the implementation of session types. This kind of approach would greatly benefit from the experience gathered in [33, 31, 30, 13].

An additional interesting result that we obtained is a very lightweight implementation of our state monad which actually makes use of the imperative functions of the underlying language. Thanks to this, and also thanks to massive inlining, we were able to obtain an average runtime performance between 5% and 15% faster than by using the .Net garbage collector and with minimal changes in the client code. This result has prompted us to start an in-depth study of all the cases where our approach is faster in order to determine its usefulness and viability.

As a side note, we believe that meta-programming (or generic programming, [39, 22, 35, 36, 18, 40, 38, 19]) might be one of the aspects that most have the potential to really make a difference when it comes to the widespread adoption of functional languages. As we have shown, meta-programming can be used not only to express very general libraries: meta-programming can also be used to optimize code, from implementing a lighter resource management system as in our case to implementing libraries that perform algorithmic optimizations (like those known from relational algebra) when doing list processing. In short, meta-programming could really be what makes it possible to build programs in functional languages that not only are safer, but also much faster thanks to the ability to meta-program optimization algorithms and various other policies.

References

[1] Robert Atkey. Parameterised notions of computation. In *In MSFP 2006: Workshop on mathematically structured functional programming, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society*, pages 31–45, 2006.

[2] David F. Bacon, Clement R. Attanasio, Han B. Lee, V.T. Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (Snowbird*, pages 92–103. ACM Press, 2001.

[3] Henry G. Baker. Cons should not cons its arguments, part ii: Cheney on the m.t.a., 1994.

[4] Henry G. Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not.*, 29:38–43, September 1994.

[5] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated - tagless staged interpreters for simpler typed languages.

[6] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2005.

[7] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[8] James Cheney and Ralf Hinze. Phantom types, 2003.

[9] T W Christopher. Reference count garbage collection. *Software Practice and Experience*, (14), 1984.

[10] Koen Claessen, Mary Sheeran, and Joel Svensson. Obsidian: Gpu programming in haskell. In *DCC*, 2008.

[11] D Clarke and A Löh. Generic haskell, specifically. In *Generic Programming. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, pages 1–4020. Kluwer Academic Publishers. ISBN, 2003.

[12] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *In ACM Symposium on Principles of Programming Languages (POPL*, pages 262–275, 1999.

[13] Iavor S. Diatchki and Mark P. Jones. Strongly typed memory areas: programming systems-level data structures in a functional language. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 72–83, New York, NY, USA, 2006. ACM.

[14] M Fähndrich and R DeLine. Adoption and focus: Practical linear types for imperative programming. Technical report, 2002.

[15] M Fluet and G Morrisett. Monadic regions. *Journal of Functional Programming*, 2006.

[16] M Fluet, G Morrisett, and A Ahmed. Linear regions are all you need. In *In Programming Languages and Systems, ESOP 2006, volume 3924 of LNCS*. Springer, 2006.

[17] Ralf Hinze. *Fun with phantom types*, pages 245–262. Palgrave Macmillan, 2003.

[18] Ralf Hinze. Generics for the masses. In *In ICFP âĂŹ04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 236–243. ACM Press, 2004.

[19] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 148–174, 2004.

[20] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 351–363, New York, NY, USA, 1986. ACM.

[21] Graham Hutton and Erik Meijer. Monadic parsing in haskell, 1993.

[22] Patrik Jansson and Johan Jeuring. Polyp - a polytypic programming language extension. In *POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[23] Wolfgang Jeltsch. Generic record combinators with static type checking. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 143–154, New York, NY, USA, 2010. ACM.

[24] Pramod G. Joisha. Compiler optimizations for nondeferred reference: counting garbage collection. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 150–161, New York, NY, USA, 2006. ACM.

[25] Pramod G. Joisha. A principled approach to nondeferred reference-counting garbage collection. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 131–140, New York, NY, USA, 2008. ACM.

[26] Mark P. Jones. Functional programming with overloading and higher-order polymorphism, 1995.

[27] Mark P. Jones. Type classes with functional dependencies. In *ESOP/ETA (LNCS)*, pages 230–244. Springer-Verlag, 2000.

[28] Richard B. Kieburtz. Taming effects with monadic typing, 1999.

[29] Oleg Kiselyov. Strongly typed heterogeneous collections. In *In Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[30] Oleg Kiselyov and Chung chieh Shan. Lightweight static capabilities, 2006.

[31] Oleg Kiselyov and Chung chieh Shan. Position: Lightweight static resources: Sexy types for embedded and systems programming. In *In TFP '07, the 8 th Symposium on Trends in Functional Programming*, 2007.

[32] Oleg Kiselyov, Simon Peyton, and Jones Chung chieh Shan. Fun with type functions version 2, 2009.

[33] Oleg Kiselyov and Chung-chieh Shan. Lightweight monadic regions. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 1–12, New York, NY, USA, 2008. ACM.

[34] Sean Lee, Vinod Grover, Manuel M. T. Chakravarty, and Gabriele Keller. Gpu kernels as data-parallel array computations in haskell, 2009.

[35] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *In ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 26–37. ACM Press, 2003.

[36] Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In *Proc. of PADL 2002*, pages 137–154. Springer-Verlag, 2002.

[37] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[38] Ian Lynagh Programming and Ian Lynagh. Template haskell: A report from the field, 2003.

[39] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Bruno C. D. S. Oliveira, Alexey Rodriguez, and Alex Gerdes. Comparing libraries for generic programming in haskell.

[40] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM.

[41] Mads Tofte and Jean-Pierre Talpin. Region-based memory management, 1997.

[42] Philip Wadler. Linear types can change the world. In *Progarmming Concepts and Methods, Sea of Galilee*, 1990.

[43] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[44] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, Proceedings of the Bastad Spring School, number 925 in Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[45] Philip Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, 1997.

[46] Philip Wadler and Peter Thiemann. The marriage of effects and monads, 1998.

[47] David Walker and Kevin Watkins. On regions and linear types (extended abstract). In *In Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 181–192. ACM Press, 2001.

[0]