Fast and safe  
resource management  
and protocol enforcement

Giuseppe Maggiore  
[maggiore@dsi.unive.it](mailto:maggiore@dsi.unive.it)

Michele Bugliesi  
[michele@dsi.unive.it](mailto:michele@dsi.unive.it)

# Abstract

In this paper we discuss how to introduce a series of powerful techniques such as reference counting and some forms of static analysis to make programs safer (and sometimes even faster) without having to build a new language. Adding reference counting is very difficult in garbage collected languages (practically all functional languages are indeed garbage collected) because there is no accessible notion of a destructor that is called when a variable exits from scope. Also, tracking protocols to ensure resources are used correctly (variables are initialized, etc.) or locking resources for cleaner multi-threaded code is something that is almost impossible to achieve statically. We discuss how through monads and quotations we can build a system that supports these constructs (and even more) in a clean, efficient and simple way.

# 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.

Modern type-safe languages are also lacking in other aspects. While classes, inheritance and in general object-orientation are an extremely expressive set of tools, still it is very verbose (if not outright impossible) to statically forbid certain sequences of operations or certain breaches of protocols. Forcing that a file is written before it goes out of scope, requiring that the same variable is not written in two different threads and other similar requirements are essentially impossible to model statically in a mainstream language such as Java or C#, even though this kind of protocol is simple, powerful and easy to describe and handle.

The solution to the first problem is, somewhat surprisingly, a solution to the second problem as well. In the remainder of the paper we will discuss how we can use the state monad (and its more powerful parametrized version) to model the scope of variables and their lifetime for reference counting. We will then show how we can track simple protocols in the usage of our resources inside the state of the state monad itself.

The structure of the paper is as follows:

* First we discuss the state monad, its syntactic sugar and its pure implementation
* Then we show how we could do simple file management where a file is closed immediately after its last reference is not accessible
* Then we discuss how we can reduce the performance overhead of the state monad with an alternate formulation of it (more suitable for an imperative language) and how we can remove some limitations of the previous implementation
* We start discussing how we can use the state parameter of the state monad, which until now we have not yet used; this allows us to track protocols that statically define the right usage of our resources
* We show how we can move back and forth between a flexible and dynamic version of our protocol tracking system and the more restrictive but much safer static version
* We describe how we could build a system of references to handle fast and safe memory allocation with no leaks and no garbage collector either
* We show how we can use our system of protocol enforcement to statically lock certain areas of memory for fast multithreaded access
* We compare our work with the existing literature
* We conclude and talk about possible extensions of our system in the future

The language we use is F#, since it allows access to most of the .Net Base Class Library and it also has very powerful support for monads. Also, since some of our constructs cannot be fully expressed in the language itself, we use quotations (a very powerful form of meta-programming) to complete our work. It is arguable (indeed we have built some working prototypes) that a language such as Haskell could achieve all this without using meta-programming facilities beyond type classes and overlapping instances. We feel that Haskell is a great tool, but it is harder to benchmark it due to its lazy evaluation scheme and its libraries are somewhat smaller than those found in commercially supported languages.

# The state monad

In functional programming, a monad is a kind of abstract data type constructor used to represent computations (instead of data in the domain model). Monads allow the programmer to chain actions together to build a pipeline, in which each action is decorated with additional processing rules provided by the monad. Programs written in functional style can make use of monads to structure procedures that include sequenced operations, or to define arbitrary control flows.

Formally, a monad is constructed by defining two operations (bind and return) and a type constructor M that must fulfill several properties to allow the correct composition of monadic functions (i.e. functions that use values from the monad as their arguments). The return operation takes a value from a plain type and puts it into a monadic container of type M. The bind operation performs the reverse process, extracting the original value from the container and passing it to the associated next function in the pipeline.

A programmer will compose monadic functions to define a data-processing pipeline. The monad acts as a framework, as it's a reusable behavior that decides the order in which the specific monadic functions in the pipeline are called, and manages all the undercover work required by the computation. The bind and return operators interleaved in the pipeline will be executed after each monadic function returns control, and will take care of the particular aspects handled by the monad.

Monads have been shown to be great assets in modern programming languages, starting from Haskell and slowly entering the mainstream through F# and C#. Monads have been successfully used to express stateful computations in pure functional languages, to add support for exceptions and exception handling, to support concurrency and even to handle list comprehensions and SQL-like queries.

One particular monad we are interested in is the state monad. The state monad allows a programmer to attach state information of any type to a calculation. Given any value type, the corresponding type in the state monad is a function which accepts a state, then outputs a new state along with a return value:

type St<'s, 't> = 's -> ('t, 's)

Note that this monad, unlike those already seen, takes a type parameter, the type of the state information. The monad operations are defined as follows:

* return produces the given value without changing the state:  
  return (x:'a) = fun s -> (x, s)
* bind modifies *m* so that it applies *f* to its result:  
  bind (m:St<'s,'a>) (f:'a-> St<'s,'b>) : St<'s,'b> =   
   fun r -> let (x, s) = m r in (f x) s

Another operation applies a state monad to a given initial state:

runState (t:St<'s, 'a>) (s:'s) : ('a \* 's) = t s

In its usual formulation the state monad is used to access the state by concatenating a series of pure functions that manipulate the state itself. This allows programs to appear imperative while still being pure. Useful state operations are those that allow to read the state, replace it and modify it in place:

get = fun s -> (s, s)

put s' = fun s -> ((), s')

modify f = fun s -> ((), f s)

## Syntactic sugar

Languages that support monads allow the programmer to use them implicitly. To do so specialized syntactic constructs are made available like Haskell *do-notation* or F# *computation expressions*. In such notations bind and return are invoked automatically. In particular in computation expressions the user defines a custom datatype (called a *builder*) with the methods *Bind* and *Return*; the user also constructs an instance of this datatype which is then used to create a computation expression. An example of a builder for the state monad could be:

type StateBuilder() = …  
let st = StateBuilder()

We now use *st* to instantiate our monad to manipulate a state made up of a pair of integers:

let get\_x = fun (x,y) -> (x,(x,y))

let get\_y = fun (x,y) -> (y,(x,y))  
let set\_x = fun v (x,y) -> ((),(v,y))

let set\_y = fun v (x,y) -> ((),(x,v))  
let program =   
 st{

let! x = get\_x  
 let! y = get\_y  
 do! set\_x (x+y)

do! set\_y (x\*y)

return ()

}

The compiler translates all the constructs that end with a bang (*!*) into explicit calls to the *Bind* method, and all uses of the keyword *return* into explicit calls to the *Return* method. Also, the variable that is introduced with a *let!* becomes the parameter of the lambda expression passed as the second parameter to the *Bind* method:

let program =

st.Bind(get\_x, fun x ->

st.Bind(get\_y, fun y ->

st.Bind(set\_x (x+y), fun () ->

st.Bind(set\_y (x\*y), fun () ->

st.Return(())))))

Executing this state monad with input *(2,3)* returns the pair *((),5,6)*.

## File Management

Let us now apply the state monad to solve the problem of automated file management. We wish to define a system that manages files and automatically closes them when they are not needed anymore. In this first use of the state monad the state parameter (*‘s*) will be *unit* (void) all the time, since we are not interested in the state that our computations work on but rather on the values that are bound and returned. This apparent artifact is only temporary (we will make heavy use of that type parameter in the next sessions) and is mainly caused by the fact that our language already supports imperative constructs and we wish to reduce the high overhead that the state monad imposes to create an imperative layer with only pure constructs; this overhead is mostly caused by the creation of a lot of nested lambda expressions, all of which will require space in the heap (memory overhead) and dynamic dispatching (runtime overhead). We will discuss optimizing the state monad further below.

We can notice an interesting fact about the state monad: whenever a variable enters in scope then it is immediately passed to the *Bind* method, so as soon as the bind method is concluded we can perform some operation to tell that variable that it has fallen out of scope; also, whenever a variable is returned then we can tell this variable that it is escaping its scope. Combining these two facts we can achieve a monad that allows us to do reference counting, thus truly eliminating the need for any informal discipline on the part of the programmer. We obtain this result in a few steps.

First we define a counting interface:

type ICounter =

abstract member incr : unit -> unit

abstract member decr : unit -> unit

Second we define a way to lift a primitive value to the *ICounter* interface; this way we can manipulate only types that implement the *ICounter* interface without any loss of generality:

type MkCounter<'a>(v:'a) =

interface ICounter with

member x.incr() = ()

member x.decr() = ()

member x.Value with get() = v

let val\_lift x = fun s -> MkCounter(x),s

we immediately lift *unit* (the resulting type will be called Unit) sincewe will need it in the definition of the monad builder:

type Unit = MkCounter<unit>

let cu = Unit()

We can now define the monad builder itself so that binding decrements the parameter as soon as its scope is ended and returning increments the parameter being returned:

type StateBuilder() =

member this.Bind<'a, 'b, 'C when 'a :> ICounter>(p:St<'C,'a>, k:'a->St<'C,'b>) : St<'C,'b> =

fun c ->

let x,c' = p c

let y,c'' = k x c'

do x.decr()

y,c''

member this.Return<'a, 'C when 'a :> ICounter>(x:'a) : St<'C,'a> =

fun c ->

do x.incr()

x,c

let st = StateBuilder()

Now we can give the first example of code that uses our monad. We define a class that implements the *ICounter* interface around a stream object. This class has two properties, *NextLine* and *EOF*, which allow access to the contents of the file line-by-line. These properties are characterized by the fact that they return values of the state monad, rather than values themselves. This is done to force all consumers of a value of type stream to remain *inside* an instance of the state monad; if this were not done a value of type stream that were allowed to escape a monad could be accessed even when deallocated. This represents a further layer of protection against accidental mistakes, and to allow this kind of pattern we need to wrap simple values such as *Bool* and *String* inside an ICounter with the *val\_lift* function:

type Stream<'C>(path) =

let mutable cnt = 1

let file = System.IO.File.OpenText(path)

interface ICounter with

member x.incr() = cnt <- cnt + 1

member x.decr() =

cnt <- cnt - 1

if cnt <= 0 then file.Dispose(); do wl("closed file " + path)

member x.NextLine with get() : St<'C,\_> = file.ReadLine() |> val\_lift

member x.EOF with get() : St<'C,\_> = file.EndOfStream |> val\_lift

We then define a constructor that creates a stream in a way that can be bound inside the monad:

let new\_stream p = fun s -> Stream p,s

At this point we can define the program itself. This program opens the file *test.txt*, reads its first line and prints it on the screen.

let program\_simple:St<unit,\_> =

st{

let! (ns:Stream<\_>) = new\_stream "test.txt"

let! nl = ns.NextLine

wl nl.Value

return cu

}

let main() =

st\_run program\_simple |> ignore

Running this program yields the expected output of:

line 1

closed file test.txt

We might wish to allow the programmer to return more complex datatypes that internally manipulate state monads. This requires to give alternate definitions to standard constructs such as tuples, union types, etc. so that they correctly implement the *ICounter* interface. One particularly interesting example arises whenever we wish to capture a *Stream* inside a closure. This means that the captured stream will now be tied to the lifetime of the closure itself and must not be deallocated before the closure, otherwise invoking the closure might result in trying to access a closed file. We define a function interface that inherits both the *ICounter* and *IDisposable* interfaces. This interface is useful if we want to support closures that can still be used outside a monad, that is closures that internally make use of monads but which can be safely used independently. While this is somewhat controversial with respect to the usual spirit of monads (using a monad must be done at the root level of a program) we believe that it is very important to support constructs that allow to break this rule safely because otherwise the constraint is excessively difficult to maintain in real life programs:

type IFun<'a,'b> =

abstract member invoke : 'a -> 'b

inherit ICounter

inherit IDisposable

We define the closure class to take as input the *ICounter* context that is captured by the closure and the function that gets passed the context, the actual input of the function and which returns the resulting state. The resulting state is required to have an empty state (the state is already passed explicitly as the context). The context is incremented and decremented in the implementation of the interfaces *ICounter* and *IDisposable*:

type Closure<'a,'b,'C when 'C :> ICounter>(ctxt:'C,f:'C->'a->St<Unit,'b>) =

interface ICounter with

member x.incr() = ctxt.incr()

member x.decr() = ctxt.decr()

interface IFun<'a,'b> with

member x.invoke v = f ctxt v cu |> fst

member x.Dispose() = ctxt.decr()

member x.st\_invoke v : St<Unit,\_> = f ctxt v

let clo c f = new Closure<\_,\_,\_>(c,f)

Now we can build our more complex example which returns a closure that manipulates the stream *ns* which is created inside the monad. Passing *ns* to the closure stops it from being disposed when it exits out of the scope of its declaration:

let program\_closure:St<unit,\_> =

st{

let! (ns:Stream<Unit>) = new\_stream "test.txt"

let print\_all =

clo ns (fun ns ->

let rec iter () =

st{

let! eof = ns.EOF

if eof.Value = false then

let! nl = ns.NextLine

do wl nl.Value

return! iter ()

}

in iter)

return print\_all

}

let main() =

use print\_all = st\_run program\_closure

(print\_all :> IFun<\_,\_>).invoke() |> ignore

The output of this program is, as expected:

line 1

line 2

line 3

last line

closed file test.txt

# A small challenge

Let us consider the following troublesome code:

# Faster and safer

The code produced until now is reasonable but has two major shortcomings.

The first problem is that it is slow because of all the nested lambda expressions; we simply cannot afford the overhead of wrapping each and every imperative statement in an explicit lambda expression because lambda expressions reside on the heap (memory overhead) and are dynamically dispatched (runtime overhead); moreover our language natively supports imperative constructs, so it makes no sense to emulate them when they are already available.

This first problem can be solved by requiring the user to code in terms of the monad, but using a definition of the state monad that makes more sense when dealing with an imperative language. We then recursively process the original source and remove all calls to bind and return, thereby yielding final code that executes as fast as the original; moreover, by defining the state monad as a value-type rather than a reference-type, manipulating values of the state monad does not require allocating memory on the heap, thereby having no impact on the garbage collector. The new definition of the state monad is simply:

type St<'C,'a> =

struct

val value:'a

new(v) = {value = v}

end

which contains exactly one value of type and where the state is never mentioned and becomes just a static annotation. We will make use of this static annotation in a later section.

We then simplify all calls to bind into their imperative equivalent (we call our transformation ):

[| st.Bind(p:St<C,A>, fun (x:A) -> k:St<C,B>) |]

(which is the de-sugarized version of let! x = p in k)  
into:

let x = state\_run [| p |]

in [| k |]

and we further simplify all calls to return:

[| (st.Return x):St<C,A> |]

(the de-sugarization of return x)  
into

St<C,A>([| x |])

Thanks to this transform we have completely eliminated from the runtime any trace of the original monad, save for some extremely computationally cheap wrapping operations in the new state monad.

The second problem lies in the requirement to pass only types that inherit from *ICounter*. We could forbid altogether passing around types that do not natively inherit *ICounter*, but this would make our system impractical; this would require us to re-create *ICounter*-friendly versions of tuples and unions, but returning *ICounter* values packed together with regular values would still be complicated and cumbersome to the final user. Alternatively we could relax this requirement a bit, allowing (as we did in the example above) to “lift” a value that is not an *ICounter* into an appropriate monadic container. This might be workable, but it would introduce various possibilities of error. Consider what would happen when lifting a value that is an *ICounter* (and thus would not require lifting) like *Stream*: inspecting the client code suggests that the stream is correctly *decr*-ed and *incr*-ed, but since it has been lifted its actual *decr* and *incr* are never really called causing possible memory leaks.

An elegant solution to this problem requires generating some code. We do so when processing calls to bind and return. Since we know that we wish to *decr* all *ICounter* values that enter a bind operation right before returning the result of the binding, and we wish to *incr* all *ICounter* values that enter a return operation, then we will simply need to modify the transform above into:

[| st.Bind(p:St<C,A>, fun (x:A) -> k:St<C,B> |]

=>

let x = state\_run [| p |]

let y = state\_run [| k |]

do decr x

St<C,B>(y)

and

[| (st.Return x):St<C,A> |]

=>

let y = [| x |]

do incr y

St<C,A>(y)

Where *incr* and *decr* are defined recursively on the structure of the type of their argument; for example (in pseudo-code):

decr (x:A) = x.decr() when A :> ICounter

decr ((x1,…,xn):(A1\*…\*An)) = decr (x1); … decr(xn)

…

decr (x:A) = () if no other match

It is worth noting that this kind of construct could be expressed very easily with Haskell-style type classes (and overlapping instances turned on) as:

class Decr a where

decr :: a -> ()

instance Decr a where

decr x = ()

instance Decr Counter(a) where

decr x = …

instance (Decr a, Decr b) => Decr (a,b) where

decr (x,y) = decr x; decr y

and so on. While we believe that Haskell is a very powerful language and its type system is clearly one of the most powerful and expressive available, its feasibility for real-world applications is somewhat limited by its relative lack of libraries and its runtime performance, which suffers a bit from the overhead of laziness. For this reason we have chosen to emulate this kind of constructs with F# quotations, which are a much lower level construct than Haskell type classes but which allow us a bigger degree of freedom.

# Annotating the state

Let us say that we want to track some protocol that defines the correct way to handle our resources. This protocol can be expressed as a series of transitions, thanks to which we ensure that certain operations happen in a certain order or according to a certain statically known scheme. Our strategy will be to store inside one type variable of the state information about current variables and their current internal state for protocol tracking. We do so by defining a linked list of static information at the level of types; we wish to represent information in the form of:

where are mutable variables and are their static information. We represent this statically with so-called *phantom types* [XXX] that will compose the state of our state monad:

type To<'x,'i> = To of 'x \* 'i

type Nil = Nil

type Cons<'hd,'tl> = Cons of 'hd \* 'tl

None of these types’ constructors will actually be ever used. We will just use these types to track static information, but at runtime they will have no function (and no performance penalty) whatsoever. State information such as:

will be represented as:

Cons<To<x1,i1>,…Cons<To<xN,iN>,Nil>…>

We extend the state information of the state monad to represent the static function represented by executing a certain statement. This means that a value of this new formulation of the state monad (also known as a *parametrized state monad*) contains two type parameters that represent how a statement’ static capabilities propagate to its predecessor:

type St<'c\_out,'c\_in,'a> = …  
let bind (p:St<'c1,'c2,'a>, k:'a->St<'c2,'c3,'b>) : St<'c1,'c3,'b> = ...

let ret (x:'a) : St<'c,'c,'a> = ...

We will consider as the static knowledge guaranteed by the execution of a statement, and as the static requirements that a statement imposes on the rest of the program in order for certain properties to be satisfied.

As an example, let us suppose that we want to make sure that all our open files are written to at least once. Then we will define types for all the files we will have and types for all the states in their protocol:

type File1 = File1

...

type FileN = FileN

type ?Empty = ?Empty

type !NotEmpty = !NotEmpty

The static information is interpreted as may be , and is interpreted as must not be . We define *new\_stream* so that it requests to the rest of the program that the file will be written:

let new\_stream<'x,'c> (path:string)

: St<'c,Cons<To<'x,!NotEmpty>,'c>,Stream<'x>> = …

Here the initial state is called *c*, and the final state is . Of course we will enforce (when processing our code) additional constraints that cannot easily (and sometimes not even at all) be expressed in the language itself. In particular with respect to *new\_stream*, we want to ensure that the name of the static variable *x* is not already bound in the state *c*:

Also notice that the *Stream* datatype is now a parametrized type; in particular, the type parameter will be the static name of the stream itself for tracking its protocol. At this point, we can define the method that writes something to a stream as:

let write\_line<'x,'c\_out,'c\_in> (s:Stream<'x>)

: St<'c\_out,'c\_in,unit> = …

The above signature is overly permissive, and so when processing our code we will add the additional constraints:

These constraints express the fact that must be in scope and accessible in both and . The *write\_line* statement relaxes the requirement from the previous statements that the file must not be empty and accepts any requirement for from the rest of the program.

*Bind* and *return* are changed too. The new signatures of *bind* and *return* are:

let bind (p:St<'c1,'c2,'a>, k:'a->St<'c2,'c3,'b>) : St<'c1,'c3,'b> = ...

let ret (x:'a) : St<'c\_in,'c\_out,'a> = ...

*Bind* simply concatenates the two static functions represented by its parameters and *return* simply creates a new statement that contains a value.

*Bind* and *return* need to have some additional static constraints. *Bind* must ensure that all variables have compatible states; for example, we want to force files to be written to at least once, that is a file must never go out of scope without having ever been written; also, bound values must be removed from the output state unless they are returned as part of the final output. We express this by stating that:

*Bind* has the additional constraint that the free static variables (those ’s found as parameters to ) that occur in the type parameter are removed from unless they also appear in . This makes sense because unless we return a stream created in *p* as part of the result of *k* then this stream will fall out of scope and will not be accessible anymore.

*Return* has the additional constraint that the free static variables found in its type parameter must be present in its state . Also, static variables that go out of scope must be permitted to do so by their protocol (that is they must be in a state that allows deletion).

## An example

Let us consider the following sample code that we wish to be able to reject:

st{

let! (f:Stream<File1>) = new\_stream "test.txt"

do! write\_line f "some line"

return ()

}

The de-sugarized version of this code, with added type annotations, is:

st.Bind((new\_stream "test.txt")

: St<Nil, Cons<To<File1,!NotEmpty>,Nil>, Stream<File1>>,

(fun f -> st.Bind(write\_line f "some\_line"

: St<Cons<To<File1,!NotEmpty>,Nil>,

Cons<To<File1,?Empty>,Nil>, unit>,

(fun () -> st.Return ()

: St<Cons<To<File1,?Empty>,Nil>, Nil, unit>))))

In this example we see that the first statement creates a new stream and adds to the requirements to the rest of the program that is not empty. The second statement satisfies this requirement and leaves a much less stressful one, that is may be left unwritten in the rest of the program. The final statement returns unit and removes from scope; this is allowed because is marked as , which is a state that allows deletion.

Let us consider another sample that we wish to reject:

st{

let! (f:Stream<File1>) = new\_stream "test.txt"

return ()

}

Its de-sugarized version is:

st.Bind((new\_stream "test.txt")

: St<Nil, Cons<To<File1,!NotEmpty>,Nil>, Stream<File1>>,

(fun f -> st.Return ()

: St<Cons<To<File1,!NotEmpty>,Nil>, Nil, unit>))))

This second version will not be accepted by our system because the state does not allow for a variable to go out of scope because this could cause a file to be closed without having ever been written to, which is exactly what we set out to avoid.

## Safely opting-in

Sometimes all these static checks may feel too restrictive, to the point that a valid program might be ruled out. To avoid this kind of scenario, we have built two different implementations of the *Stream* datatype. The first is the first one we have seen, where there are no static checks of any kind; we add dynamic checks to this implementation so that its operations *may* fail if they break our protocol (that is the file gets closed before anything gets written to it). The second implementation is the one we have just seen with static checks. We give two operations for “forgetting” static information and guessing it back:

let to\_unchecked (s:Stream<x>) : St<'c\_out,'c\_in,Stream> = …

with the additional constraint that , and:

let to\_checked<'x,'i,'c\_out,'c\_in> (s:Stream) : Option<St<'c\_out,'c\_in, Stream<x>>> = …

where we guess that the internal, dynamic state of parameter *s* is , thereby requiring . If this is the case then we get an appropriate output state, otherwise we get nothing (the *None* constructor of the *Option* type). This way an application may fall back to a dynamic version which may fail at runtime by turning a *Stream<x>* into a simple *Stream* whenever it needs the extra flexibility and it could “downcast” to a *Stream<x>* by asserting a particular internal state of the stream in question.

# References and heaps

A very natural application of our system is to handle pooled memory locations. In particular, given a program that allocates a very large number of instances of a particular class, but which uses them for a short time before leaving them to the garbage collector, the performance (and occupied memory) penalty associated with memory management can be surprisingly high. In this case it might make sense to store many instances of pre-allocated values and pass around references to these values. These references act as lightweight proxies for the objects they store and we can even use them for reference counting. This is quite important since it frees the user from the burden of manual deallocation that is traditionally associated with memory (and in general resource) pooling.

A heap is defined as some container that stores a set of values of a given type , together with some information that tells which one of these are used by some active reference and which one of these are free. Given a heap we can create an appropriate reference to a free element in the heap:

type Heap<'a> = ...

let create\_new (h:Heap<'a>) : St<'c,'c,Reference<'a>> = ...

The heap allows for reference counting, that is each location stores the number of references that are actively pointing to that cell.

A reference contains the index of the element it points to, and allows us to access the pointed element and assign to its location inside the heap:

type Reference<'a> = ...

let (!) (r:Reference<'a>) : 'a = ...

let (:=) (r:Reference<'a>) (v:'a) : St<'c,'c,unit> = ...

A reference also implements the *ICounter* so that the pointed cell will be freed as soon as all references to it fall out of scope.

One relevant note is that a simple implementation of reference counting risks encountering problems in the presence of circular references. For this reason either we leave circular data structures to the garbage collector or we add more complexity to our implementation.

## Preliminary benchmark

Preliminary tests show that a snippet of code taken from an actual application can obtain a speedup, thanks to the garbage collector which does not get invoked anymore, of up to . The sample code is a game main loop where at each iteration a projectile may be shot or a ship may be destroyed; since this code is run at a high frame-rate, between 30 and 60 times per second, it is very clear how this kind of garbage collection overhead could have a noticeable impact.

# Locking heaps

As we did for files, we can give a static name to heaps in order for them to be manipulated statically. We also define the static flags , and to express restrictions on a heap and all its references. The heap datatype changes slightly into:

type Heap<'a,'x> = ...

since now the heap also stores its static name . Creating a heap modifies the state, so its creation function manipulates the state:

let new\_heap<'a,'x>() : St<'c\_out,'c\_in,Heap<'a,'x>> = …

With the additional constraint that and . References also store the static name of the heap that stores them:

type Reference<'a,'x> = ...

let create\_new (h:Heap<'a,'x>) : St<'c,'c,Reference<'a,'x>> = ...

Manipulating a reference has some constraints on the state:

let (!) (r:Reference<'a,'x>) : St<'c,'c,'a> = ...

let (:=) (r:Reference<'a,'x>) (v:'a) : St<'c,'c,unit> = ...

Dereferencing with (!) requires that , that is the heap associated with is neither nor . Assigning with (:=) requires that , that is the heap associated with is neither nor .

At this point we can use these facilities to create multi-threaded programs that do not need locks in order to avoid dangerous operations such as write-write in different thread, since we can statically lock an entire heap on a thread thereby inhibiting certain dangerous operations on it from the thread in question:

let fork (t1:unit -> St<'c\_out1,'c\_in,'a>) (t2:unit -> St<'c\_out2,'c\_in,'a>)

: St<'c\_out,'c\_in,'a \* 'b>

We add to *fork* the constraint that all heaps accessible in must still be accessible in both and but with compatible capabilities, that is the same heap must not be writable in both threads.

## More preliminary benchmarks

Further preliminary tests show that when we process the entities of two large heaps in two different threads without actually locking (but with the guarantee that no nasty write-write conflicts will arise) then this program is up to faster than its identical counterpart which uses locks to ensure its correctness. While this experience is somewhat anecdotal and we will definitely test our system on a much larger set of samples, it is worth pointing out that the tested snippet was not chosen for any particular reason but rather it was quite a random choice.

# Related Works

## Region-based memory management.

Tofte and Talpin 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 extends this work by allowing explicit region allocation and deletes, while making sure that all data accesses to a region happen during its lifetime. Similarly, Niss and Henglein study an explicit region calculus, albeit for ﬁrst order programs. 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.

Linear type systems

Starting with Wadler, 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 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.

**Lighweight static capabilities**

Static capabilities have been implemented by Kiselyov et Al. 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.

**Lightweight Monadic Regions**

Kiselyov et Al. 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.

## Strongly Typed Memory Areas

Jones et Al. 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 deﬁne 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 speciﬁcations and standards that must be carefully followed by application programmers and language implementations.

# 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 (object orientation and garbage collection as the most prominent examples) solve many problems and affords greater expressivity, but they have a cost in terms of performance and cannot capture many interesting 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, some of them require further work. When building our system 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 in literally supporting full-sized embedded languages with powerful static analysis in commercial languages such as F#.

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, from abstract interpretation to simple state machines that govern the behavior of certain entities up to the implementation of session types. On the other hand, it would greatly make sense if we could make our meta-programming library much more parameterized. This would be interesting for emulating type-classes and from very early tests it looks like it could be achieved with a monad, as seen in various uses of monads for expressing logical constraints.

In conclusion, we believe that modern languages are getting powerful enough to express many of the extremely interesting constructs that have been studied until now, but that have obtained little or no adoption in practice. This also shows that for building very smart libraries the most advanced features of functional languages find one of their best, practically useful realization.

# References

Monadi

Phantom types

Type functions

Qualified types

Regions

Capabilities

Monadic regions

Lightweight monadic regions

Finally tagless, partially evaluated (rappresentazione di linguaggi embedded)

Adoption and focus

Lightweight static capabilities

Static when possible, dynamic when needed

Strongly typed memory areas

Lightweight static resources

Qualified types for ML

Linear types

Generic record combinators

HList