Monadic Reference Counting

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.

[SIMIL-HASKELL]

# The state monad

[PRESENTAZIONE DELLA STATE MONAD]

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 = λs

let y,s' = p s

in k y s'

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

return :: a St s a

return x = λs x,s

## Syntactic sugar

[DO NOTATION]

# 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, 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 and a type . is the type of our actual resource, and is the proxy that we will use to access this resource. The proxy may:

* be created from a value with
* increment its internal counter with
* decrement its internal counter with
* get its internal value with
* get its internal counter with

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

## 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 after being created with
* returns a value with when its count is greater than ; otherwise returns
* and respectively increment and decrement the internal count by , but only if the count is greater than ; otherwise they both return

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

s x . (do r ← new x

count r) s = (1,\_)

s r . (get r) s = (\_,\_) (count r s) = (c,\_) c > 0

s r . (do incr r

count r) s = (c+1,\_) (count r s) = (c,\_) c > 0

s r . (do decr r

count r) s = (c-1,\_) (count r s) = (c,\_) c > 0

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

## Reference implementation

Our definition of Reference may appear a bit excessive. What is the meaning of the functor ? is the type of references, which will be used as proxies of the actual values of type . 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)]  
 …

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 into a value and its associated counter:

instance (Map s, k ~ Key s) ⇒ Reference k a (Map k (Int,a))  
 …

where means that is a map and is a type function that returns the type of key used to index elements of ; 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)|]|]  
 …

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 ):

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 and functions:

class Reference f a s where

new :: a -> St s (f a)

incr\_step :: f a -> St s ()

decr\_step :: 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 datatype is the witness of the isomorphism between two type and

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

As an example, 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 :

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 *incr\_rep* and decr\_rep that recursively invoke themselves in order to invoke *incr\_step* and *decr\_step* respectively on each Reference found inside the original Reference:

incr\_rep :: Reference f a s => Rep a -> (f a) -> St s ()

incr\_rep (RUnit) ref = incr\_step ref

incr\_rep (RSum ra rb) ref =

do v <- get ref

case v of

| Inl x -> incr\_rep ra x

| Inr x -> incr\_rep rb x

incr\_step ref

incr\_rep (RProd ra rb) ref =

do (x,y) <- get ref

incr\_rep ra x

incr\_rep rb x

incr\_step ref

incr\_rep (RType ra ep) ref =

do v <- get ref

incr\_rep ra (from ep v)

incr\_step ref

decr\_rep :: Reference f a s => Rep a -> (f a) -> St s ()

…

We omit the body of *decr\_rep* since it is substantially identical to that of *incr\_rep*, 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

incr\_step = return ()

decr\_step = 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 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 and functions:

incr :: Reference f a s => f a -> St s ()

incr ref = incr\_rep rep ref

decr :: Reference f a s => f a -> St s ()

decr ref = decr\_rep rep ref

## 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 = λs →

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 = λs → 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.

# 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.

## Reference example

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 and are completed, then we may not release any memory *at all*! This is clearly nonsense, since can be released right after the call to and can be released right after the first recursive call to .

## Explicit continuations

Enter Continuation *Returning* Style. We try and solve this problem with trampolines, 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:

Trmp s a = s -> ((s->(a,s)),s)

1

2

3

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 = λs → (λs →

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:

return :: Reference f a s ⇒ f a → Trmp s (f a)

return x = λs→(λs → 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 = λs →

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 much sooner than the standard binding, since get decremented *before* the full execution of . The idea is that has a chance to increment as its , and right after this has been done returns . The actual execution of is thus stored as , which may contain recursive calls to some caller. Before executing this (possibly time-consuming) code we get our chance to free in case it were not needed anymore inside :

(>>=) :: Reference f a s ⇒ St s (f a) → (f a → Trmp s b) → Trmp s b

p >>= k = λs →

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.

## Binary tree solution

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 (λpts → return mk\_leaf pts)

else

trmp pts (λpts →

do m ← median pts

trmp (pts,m) (λ(pts,m) →

do l,r ← split pts m

trmp (l,r) (λ(l,r) →

do tl ← bt l

trmp (tl,r) (λ(tl,r) →

do tr ← bt r

trmp (tl,tr) (λ(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 [REFERENCE NEEDED] world-passing-style of the state monad, but we are also passing around the active scope.

## 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 keyword (not to be confused with the private function seen above).

The translation rule is quite straightforward:

[| trmp x ← m

n |] = trmp ctxt (λctxt → m >>= fun x → [| trmp n|])

ctxt = FC(m)FC(n)

[| trmp return m |] = trmp ctxt (λctxt → return m)

ctxt = FC(m)

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.

# Parametrized monad and state

We now move to a more powerful definition of the state monad, the parametrized state monad. 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 point to the same type of storage (lists, lists of lists, maps, arrays, etc. as seen in [Reference implementation](#_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 :: ( ~ Storage f a, Addable s, .+ s) ⇒ a → St s ( .+ s) (f a)

incr :: ( ~ Storage f a, s) ⇒ f a → s → s

decr :: ( ~ Storage f a, s) ⇒ f a → s → s

get :: ( ~ Storage f a, s) ⇒ f a → St s s a

count :: ( ~ Storage f a, s) ⇒ f a → St s s Int

In particular 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 . The 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 is the operator. The added item must be available in the resulting type, and to ensure this we use the predicate. and 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 and .

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 :

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 type class. A type can be added to a type (the result is ) if it respects the predicate:

class Addable x s where

s .+ x :: \*

add :: s → s .+ x

A type is part of the heterogeneous list 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 = λ(x .: s) (f x) .: s

instance x s x y .: s

lift f = λ(y .: s) y .: (lift f s)

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

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 inside the final state, which in our case has type . For this reason we require that , so that we will be able to the decrementing operation from into :

(>>=) :: (Reference f a, Storage f a r) ⇒ St p q (f a) → (f a → St q r b) → St p r b

s >>= k = λp →

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.

# 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

[PRELIMINARY BENCHMARKS]

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