Resource Polymorphism

Guillaume Munch-Maccagnoni (Inria, LS2N CNRS)
Guillaume.Munch-Maccagnoni@Inria.fr

6th March 2018

We present a resource-management model for ML-style programming languages, designed to be compatible with the OCaml philosophy and runtime model. This is a proposal to extend the OCaml language with destructors, move semantics, and resource polymorphism, to improve its safety, efficiency, interoperability, and expressiveness. It builds on the ownership-and-borrowing models of systems programming languages (Cyclone, C++11, Rust) and on linear types in functional programming (Linear Lisp, Clean, Alms). It continues a synthesis of resources from systems programming and resources in linear logic initiated by Baker.

It is a combination of many known and some new ideas. On the novel side, it highlights the good mathematical structure of Stroustrup's *"Resource acquisition is initialisation"* (RAII) idiom for resource management based on destructors, a notion sometimes confused with finalizers, and builds on it a notion of resource polymorphism, inspired by polarisation in proof theory, that mixes C++'s RAII and a tracing garbage collector (GC). In particular, it proposes to identify the types of GCed values with types with trivial destructor: from this definition it deduces a model in which GC is the default allocation mode, and where GCed values can be used without restriction both in owning and borrowing contexts.

The proposal targets a new spot in the design space, with an automatic and predictable resource-management model, at the same time based on lightweight and expressive language abstractions. It is backwards-compatible: current code is expected to run with the same performance, the new abstractions fully combine with the current ones, and it supports a resource-polymorphic extension of libraries. It does so with only a few additions to the runtime, and it integrates with the current GC implementation. It is also compatible with the upcoming multicore extension, and suggests that the Rust model for eliminating data-races applies.

Interesting questions arise for a safe and practical type system, many of which have already been thoroughly investigated in the languages and prototypes Cyclone, Rust, and Alms.

Contents

1 Introduction

A resource is a value that is hard to copy or dispose of. A typical resource is a large data structure, a file handle, a socket, a lock, a value from a cache, an exclusive access to a mutable, a value originating from a different runtime, or a continuation. It is also any data structure composed from such resources, such as a list of resources or a closure containing a resource. Support for resource management in programming languages (PLs) is a concern for safety, efficiency, interoperability and expressiveness.

This is a proposal for a resource-management model compatible in broad strokes with the OCaml[1] philosophy and runtime model. By abstracting a few low-level technical details, it can also be read more generally as a model for other languages in the ML family.

It considers new values and types that own or borrow resources, similar to ownership/borrowing in C++11[2]/Rust[3], in addition to the current GC types. It is motivated by addressing issues that arose during discussions with several Serious Industrial OCaml Users concerning safety, efficiency, interoperability, and expressiveness in the presence of resources: the inadequacy of

[1]https://ocaml.org/
[2]https://isocpp.org/
[3]https://www.rust-lang.org/

finalizers for timely disposal of large pools of resources, the unpredictability and limitations of unboxing, the difficulty of interfacing with resource-intensive libraries such as Qt[4], the difficulty of cleaning-up resources reliably with fibers in ocaml-multicore[5], the need for affine closures with effect handlers, and more.

By OCaml philosophy, what is meant is reaching a sweet spot combining:

1. A safe type system that helps instead of hindering,
2. Lightweight and expressive abstractions,
3. An efficient runtime.

This proposal focuses on the levels 2 and 3 above. It is voluntarily vague about level 1, for the reason that resource-friendliness is deeply rooted in the computational behaviour (i.e. level 3), as it will become clear. Let us put aside the idea that one can start from ideas for a type system and expect interesting computational behaviour to suddenly appear; instead, the type system should come in a second time, in service of a convincing design for computational aspects. The first challenge for OCaml, tackled here, is to get level 2 right, such that level 3 can become a realistic, conservative, and useful extension of the current runtime. Level 1 is expected to require substantial effort; hopefully the proposal provides sufficient motivations for such an effort. Besides, we will see that there is ample prior work addressing various questions for a practical type system, already at work separately in the languages Rust and Alms.

The model is inspired by Stroustrup's RAII (Stroustrup, 1994) and Hinnant et al.'s move semantics in C++11 (Hinnant, Dimov, and Abrahams, 2002). RAII (*"Resource acquisition is initialisation"*) proposes to integrate error handling and resource management by attaching destructors to types: clean-up functions that are called automatically and predictably when a scope ends, whether by returning or due to an exception being raised. It is an essential ingredient in the *basic exception-safety guarantee* (Stroustrup, 2001) which requires that functions that raise an exception do not leak resources and leave all data in a valid state. In the words of Ramananandro, Dos Reis, and Leroy (2012), RAII enforces invariants about the construction and the destruction of resources predictably and reliably.

In contrast, *finalizers* as currently used in OCaml, that is clean-up functions called by the garbage collector, are not predictable, are not guaranteed to be run, and allow making values reachable again (Minsky, Madhavapeddy, and Hickey, 2013, Chapter 21; similar points are made for finalizers in other languages in Stroustrup, Sutter, and Dos Reis, 2015). In OCaml, which thread calls the finalizer is even explicitly unspecified[6]. Finalizers appear commonly considered inappropriate for managing resources.

In addition, while the original resource-management model of C++ was criticised for its over-reliance on deeply copying values (among others), Hinnant et al. proposed to introduce a new kind of types in C++ (*rvalue references*) that allowed to express the moving of resources. In particular:

[4]https://www.qt.io/
[5]https://github.com/ocamllabs/ocaml-multicore
[6]http://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html

- it supported a movable and non-copiable pointer for automatic resource management (*unique_ptr*), reminiscent of uniqueness (Barendsen and Smetsers, 1996) and ownership types (Clarke, Potter, and Noble, 1998),

- it supported a polymorphism of resource management: the management of a type is by default deduced from its components,

- it supported the conservative extension of data structures and algorithms from the standard library to operate on resources,

- all the while retaining backwards compatibility with existing code (sometimes even speeding it up by removing unnecessary copies).

Together with the extensive use of (unsafe) passing by reference and a rudimentary form of reference-counting garbage collection expressible with RAII (*shared_ptr*), this is advocated as the new resource-management model of C++11 (Stroustrup, Sutter, and Dos Reis, 2015). Ownership types, and regions as in MLKit (Tofte and Birkedal, 1998) and Cyclone (Jim, Morrisett, Grossman, Hicks, Cheney, and Wang, 2002; Grossman, Morrisett, Jim, Hicks, Wang, and Cheney, 2002), have also been proposed as abstractions amenable to static analyses, which has inspired Rust's ownership-and-borrowing model strengthening the C++11 model with static safety guarantees and a novel design for preventing data races (Anderson, Bergstrom, Goregaokar, Matthews, McAllister, Moffitt, and Sapin, 2016).

This proposal, however, should not be seen as just trying to extend OCaml with C++ idioms. Its starting point was the similarities between C++'s resource polymorphism and polarisation in proof theory, as well as a rational reconstruction of destructors in the linear call-by-push-value categorical model (Curien, Fiore, and Munch-Maccagnoni, 2016). This suggested several aspects of this proposal, by bringing to light the deep compatibility of the C++11/Rust resource-management model with functional programming. This continues a synthesis of systems programming's resources and linear logic's resources initiated in a series of rarely-mentioned articles by Baker (1994a,b, 1995).

In the end, polishing RAII brings additional similarities with Rust, a runtime model that fits that of OCaml, and applications that go beyond a simple replacement of finalizers. This proposal is an element in a broader thesis that RAII hides a fundamental computational structure that has not been given yet the exposure it deserves.

1.1 The relevance of C++11 for OCaml

The proposal will remind of linear types, regions, uniqueness types, ownership types, and borrowing, *à la* Linear Lisp, Clean, Cyclone, Rust, etc. (Section 13 offers a more detailed comparison.) But in comparison to Linear Lisp, Clean, Cyclone, Rust, etc., the experience of the move from C++98 to C++11 stands out for OCaml for three reasons:

- Both are established languages that need to preserve the meaning and performance of large amounts of legacy code.

- Both have to deal with exceptions, a core part of their design, much more prominently than in Rust in which exceptions (*panic*) are restrictive and discouraged by design, or than in other languages in which they are absent.
- Both are designed around light, efficient, and predictable abstractions.

Of course, C++11 and OCaml differ greatly in other technical and principal aspects, and their communities do not overlap much, which would explain why, if there is any value to this proposal, it has not been proposed before, besides the example set by Rust. In addition, some experience with mathematical models of PLs in the categorical tradition has helped (the author at least) reading between the lines of the C++11 specification and of various idioms that arose from it, and extracting its substance. The inputs from semantics are explained in the next section.

While the crucial ideas for this proposal are RAII and move semantics from Stroustrup and Hinnant et al., the end result is probably closer to Rust. This is because Rust itself was inspired by both C++ and ML among others (Anderson et al., 2016). Compared to C++, Rust offered language support for isolating the *"unsafe"* parts of the code in libraries. This means that in most of user code, what are merely *best practices* in C++11 are enforced by the Rust compiler, thereby providing strong static safety guarantees. Moreover, by treating mutable state as a resource, Rust tracks aliasing, ensuring that no data races are due to bugs in *"safe"* mode. This tour de force for an industrial language drew the attention of the academic PL community. C++ also takes example on Rust with the ongoing *Core Guidelines initiative* (Stroustrup and Sutter, 2015) aimed at standardizing and tooling a *"smaller, simpler, safer language"*, with similar ideas as Rust but an emphasis on easy migration of legacy code.

This proposal suggests that a similar path is possible for OCaml, where *"small, simple"* (and efficient) is retained from the OCaml that everyone likes, and where *"safer"* is achieved for resources and concurrency, without sacrificing the expressiveness of a functional, GCed language. This proposal focuses on resource safety; Rust also tries to solve the problem of data races. In the current proposal, OCaml's unrestricted shared mutable state is kept for the sake of backwards compatibility. Proposing a solution to data races in OCaml *à la* Rust in one go would be ambitious. The requirement of backwards compatibility can seem a convenient excuse to not propose one right away, but in fact it shows an opportunity to first integrate some language features that are already useful for resource management, while at the same time providing a richer playground for tackling concurrency problems in the future. Nevertheless, Section 13.3 comes back on this suggestion on an optimistic note.

1.2 RAII, resource polymorphism, and GC, from a semantic point of view

RAII is an idiom in which a destructor is attached to a resource, according to its type. The destructor is called predictably when the variable goes out of scope, including due to an exception being raised. With RAII, one can allocate resources on the heap, and have the resources automatically collected predictably and reliably, bypassing garbage collection. The destructor deallocates memory, and can be customised by the language or the user to perform other clean-up duties.

Three intuitions arising from mathematical models were the starting points of this proposal:

1. Hinnant et al.'s resource polymorphism for C++11 coincides with Girard's polarity tables in proof theory, which describes compound connectives in terms of basic ones (Girard,

1991, 1993). In proof theory, the goal of polarity tables is to minimize the number of modalities, so as to maximize the number of type isomorphisms (Girard, 1991) and "η" conversions (Danos, Joinet, and Schellinx, 1997); for PLs these concerns have an immediate application in making programs easier to reason about. (Girard 1993 describes the mixing of linear, intuitionistic, and classical logics in the same system, which in fact corresponds to a mixing of resources and continuations.) For the purposes of this proposal, a polarity is a type of types closed under constructions. This view is close to that of *"kinds as calling convention"* (Bolingbroke and Peyton Jones, 2009; Eisenberg and Peyton Jones, 2017), itself inspired in part by Levy (Bolingbroke and Peyton Jones).

Both Girard and Hinnant et al. describe systems in which, instead of a non-copyable pair $\otimes$ distinct from a copyable one (maybe written differently, $\times$), there is a single pair $\otimes$, whose polarity (e.g. whether it is copyable or not) is deduced from its components: $A \otimes B$ is copyable by default as soon as both A and B are; in this case the copy operation consists in the sequence of copies of members, which turns out to have an abstract description in the form of a canonical algebraic structure (Bierman, 1995). (With linearity interpreted as counting uses, a similar idea was proposed in some form by the Clean language, and it has reappeared several times. See Section 3.)

2. We have noticed, in joint work with Guillaume Combette, that the notion of scoped-tied destructors arises naturally when modelling exceptions and local control (return, break...) in the linear call-by-push-value (LCBPV) model of effects and resources. CBPV (Levy, 1999, 2004) can be seen as an idealised model of ML-style languages (higher-order, typed, with strict data types and effects) refining the call-by-value λ-calculus, and LCBPV is a natural decomposition of it generalizing linear logic (Girard, 1987) with hopes that it could serve to model the interaction of effects and resources. Nothing prefigured that the model had to do with idioms from a systems programming language.

   The lesson is that trying to model exceptions or local control in LCBPV naturally leads to the rediscovery of scope-based destructors, and of several of their peculiarities which are described below. It provides the perspective that the interpretation of resources as affine types is not at odds with linear logic, but in fact arises from it. In contrast, LCBPV does not justify other resource-management idioms: it attributes no meaning to finalizers, and *try...finally* (e.g. Java) is as *ad-hoc* as one expects. This suggests that RAII is a fundamental computing concept, similarly to CPS given that it arises from the same kind of algebraic considerations.

3. While the previous points suggest that a practical resource-management model has to mix different resource-management techniques as determined by the type, it still remains to explain how an ownership-and-borrowing model can be integrated in a language with a GC. For this, let us draw from the consensus that the GC should not have to perform non-trivial finalization, and make it a definitional principle:

   > the GC is a run-time optimisation that either delays or anticipates the collection of values that can be trivially disposed of.

From this definition, we will derive a way of mixing GCed values with resources, alternative to using finalizers.

In any case, there is a leap of faith between the linear call-by-push-value models as they currently stand and the current proposal. It would be pointless to get into more mathematical details at this point. Then, what is the value of the abstract point of view? To begin with, a contribution of the semantic view on RAII is to reassure about various peculiarities which might otherwise seem *ad-hoc*:

- It gives rise to a notion of affine types that, instead of looking at odds with the linear logic narrative (why affine rather than linear?), arises naturally from it, and happens to match idioms used in successful industrial languages.
- It sheds a natural light on complicated rules, such as the rules for automatic generation of destructors (for instance in C++ if two types A and B have destructors d_A and d_B then $A \otimes B$ has destructor $d_A \otimes d_B$ which performs d_A and d_B in sequence). Such rules actually describe a canonical mathematical construction and therefore enjoy good properties not necessarily visible from the surface of a language.
- It attributes no meaning to exceptions escaping from destructors. In C++ too, this is undefined behaviour: otherwise one can end up with several exceptions being raised at the same time if destructors throw during stack unwinding.
- It explains peculiarities of pattern-matching in the presence of ownership: the common explanation in terms of universal properties seems to recover the intuitive fact that assuming ownership during pattern matching is only possible if the destructor of the pattern is the default one. This predicts an integration of ownership with pattern matching already explored by Rust.

The value of the model is therefore to encourage a bold change, to which one would not necessarily come in small steps by trying to fix separately the various issues related to resource management in OCaml. We hope to explain the (modest) mathematical aspects in greater detail elsewhere.

Outline

Section 2 describes the integration of ownership and borrowing with a traced garbage collector based on a notion of resource polymorphism. Section 3 come back on move semantics and resource polymorphism from a historical perspective. From Section 4 to Section 12, the proposed new abstractions are described and examples using them are given. Implementations of the examples are given in current OCaml through a whole-program translation, that clarifies the computational meaning of the abstractions and highlights the current limitations of the runtime and type system. In Section 13, the proposal is compared to existing PLs.

Acknowledgements

Thanks to Raphaël Proust for introducing me to the topic of resource management in PLs, and to Leo White for freely sharing thoughts on the situation in OCaml. Thanks to Guillaume Combette: a milestone for this proposal was an elementary reconstruction of RAII in LCBPV which we obtained during his visit at LS2N, and which motivated several aspects of it. Thanks to Frédéric Bour, Thomas Braibant, and François Pottier for freely sharing their interest and experience with this topic. Thanks to Gaëtan Gilbert, Adrien Guatto, Jacques-Henri Jourdan, Gabriel Scherer, and Leo White, for comments and questions about an earlier version of this proposal.

2 Integrating ownership and borrowing with a GC

Let us now describe the integration of a GC in an ownership-and-borrowing system *à la* C++/Cyclone/Rust following the definition of the GC as a runtime optimisation for the collection of trivially destructible values. (For the moment the unboxing optimisations are not considered, but they do not fundamentally change the model, see Section 12.2.)

2.1 Owned, borrowed and GCed values

There are GCed values typed with GC types, which never have a destructor. They can be passed and returned freely. These are those already present in OCaml.

Let us add types for resources, which are not managed by the GC but with RAII, and which do have destructors that are called in a predictable fashion. A value of the latter type is *owned*. Owned values can be moved, which transfers ownership, for instance to the caller, to a callee, or to a data structure. This means that responsibility for calling the destructor is transferred along with it. *Ownership types* combine in order to form other ownership types (for instance a list of ownership type is an ownership type). Static analysis ensures that a moved value is no longer accessed by the previous owner, for instance with an affine type system.

In addition, owned values can be *borrowed*. A borrowed value is a copy which is given a *borrow type*, denoting that the responsibility of calling the destructor belongs to somebody else. Borrow types combine to form other borrow types. There is no restriction on the amount of times a borrowed value can be passed, but it should not be accessed after the original value has been disposed of. This can be ensured by a static analysis (typically inspired by type-and-effect systems as introduced in Tofte and Talpin, 1994); practical static analyses combining this idea with ownership/linear types have already been experimented in Cyclone and Rust (see in particular Fluet, Morrisett, and Ahmed, 2006). Thus borrow types will need to carry annotations similar to Rust's lifetimes.

There are now three modes of management:

(G) **GCed values** and **GC types**

(O) **Owned values** and **ownership types**

(B) **Borrowed values** and **borrow types**

Each have pros and cons:

- GCed values can be copied freely, but cannot have destructors.
- Owned values can only be moved, which allow them to support destructors, to be used in a producer/consumer interaction (such as between components of a program, or to receive and pass values from/to other runtimes), to denote uniqueness, and to deal with large structures without impacting the cost of tracing.
- Borrowed values can be copied, but subject to the restriction that it does not outlive the resource it originates from.

Such a diversity is not realistic without a plausible notion of resource polymorphism:

- for the conciseness of the language,
- for the expressiveness when mixing GC types and non-GC types,
- at the level of types and their meaning, for simplicity and clarity for the user, and
- at runtime, for a simple and efficient implementation.

The example of C++11 shows that such a notion of resource polymorphism also helps for backwards compatibility and extensibility of libraries.

The core part of the design is to understand GCed values polymorphically both as borrowed values and as owned values.

2.2 Resource polymorphism and runtime representation (level 3)

RAII is a notion tightly integrated into the runtime. Let us start there. In addition to traced pointers (with lowest bit set to 0), let us use untraced pointers (with lowest bit set to 1). The latter are allocated in the major heap and deallocated using RAII. An untraced pointer can either be borrowed or owned. If borrowed, there is nothing to deallocate. If owned, a compiler-generated destructor is called at the end of the scope, which deallocates the memory.

The following invariant is maintained throughout: any live GCed value is reachable either from the stack/registers, or from registered roots. In order to use a GCed value as a sub-value of an owned value, the GCed value is registered as a root at allocation. The compiler-generated destructor is then in charge of unregistering the root. Thus RAII is essential for the absence of leaks in the presence of exceptions. If the sub-value is in the minor heap, the new pointer is registered as a major-to-minor pointer and treated as such.

This leads to a first table for combining G, O, B types according to whether the resulting value is allocated by GC or not (in order: the strict pair, the type of lists, and the type of borrows, the latter of which is an addition of the proposal):

`X`	`X*G`	`X*O`	`X*B`	`X list`	`&X`
`G`	0	1	0	0	0
`O`	1	1	1	1	1
`B`	0	1	0	0	x

0: traced pointer (lowest bit set to 0)
1: untraced pointer (lowest bit set to 1)
x: either, same as original

Table 1: Runtime representation for newly-created values in function of the polarity

In words:

- Structures comprised of owned values are allocated with RAII,
- Discarding borrowed values is trivial, so structures comprised of borrowed values and/or GCed values are allocated with the GC,
- Any combination of GCed values and owned values is owned; RAII is used to register and unregister the GCed value as a root as explained above.

Thus, the runtime representation of structures alternates GCed phases and non-GCed phases: a GCed value can contain a non-GCed value by borrowing, and a non-GCed value can contain GCed values by rooting. Notice that without borrowing, the heap would have a much simpler structure consisting of a RAII phase with leaves pointing to a GCed phase. From this angle it is clear that borrowing is essential for expressiveness.

Passing (*copying*) a borrowed or GCed value is done by copying the pointer. Owned values cannot be copied; instead, passing (*moving*) an owned value involves copying the pointer and setting the original to zero. Recording the move by setting the pointer to zero is required because destructor calls are determined statically, and since resources are affine, it cannot be known statically whether a resource has moved: for instance there can be branching code paths in which only one path moves the resource. Destructors therefore need to know at runtime whether a resource has moved. Of course, this imposes a linear (affine) treatment of owned values.

Thus, for each type, the compiler generates a destructor (whose representation can be defunctionalised), which: 1) tests for zero to detect whether the variable has moved, 2) if not, applies user-supplied destructors, and 3) deallocates. On the back-end, modular implicits (or at least their back-end) can be re-used for generating the destructor, which in turn allows the proposal to scale to abstract types and polymorphic functions: functions polymorphic in an Ownership type variable take an implicit module as an argument, which contains the compiler-generated destructor.

Reference counting can be used for optimising the case where several roots point to the same value. However, this is not the form of reference counting that has been criticised for garbage collection. It avoids the well-known drawbacks of reference counting: the difficulty to collect cycles, the up-front cost, and cascades of reference count updates. Indeed, the collection is still ultimately performed by tracing, and the cascades are avoided for two reasons:

1. only the pointer at the interface between the RAII phase and the GCed phase is reference-counted, and
2. copying roots can only happen by borrowing, and in this case it is not necessary to update the reference count since the reference does not outlive its resource.

All in all, GCed values are determined to be reachable by a mix of tracing and reference counting. There was a definite influence in the design of this proposal from the thesis that all GCs lie in a spectrum between tracing and reference counting (Bacon, Cheng, and Rajan, 2004).

2.3 Resource polymorphism, types, and meaning (level 2)

Let us now provide an explanation to the runtime model in terms of types.

A GCed value is typed by a GC type, an owned value by an ownership type and a borrowed value by a borrow type. The mode of resource management (or polarity) of a type is determined by induction according to the following polarity table:

X	X*G	X*O	X*B	X list	&X
G	G	O	B	G	G
O	O	O	O+B	O	B
B	B	O+B	B	B	B

where:

- `O` signifies linearity restrictions,
- `B` signifies lifetime restrictions,
- `G` signifies no restrictions.

Table 2: Usage restrictions in function of the polarity

In words:

- The modes G, O, B are closed under constructions.
- A GC type can be used to form both ownership types (in combination with ownership types) and borrow types (in combination with borrow types).
- The type of borrows is a borrow type, except for the type of borrows to a GC type which is a GC type (itself, obviously).

In other words, a GC type can be seen both as an ownership type and a borrow type. Indeed:

- *A GCed value is owned* in the sense that holding the pointer (i.e. copying it to the stack) is sufficient to prolong the life of the value. Moreover, GCed values can be moved in the same way as owned values are moved. The fact that GCed values can be used like owned values can therefore be reflected in the language, so that it is possible to give a single resource-polymorphic implementation to the function which takes two lists as argument and merges them, for instance.

- *A GC type can also be seen as a borrow type*, in the sense that GCed values can be copied without restriction. There is no difference between a GCed value and a resource with trivial destructor allocated at the largest region of the program, if the GC is considered as an optimisation anticipating its collection.

For instance, a GCed value can be extracted from inside a borrowed value and passed in an owned context, prolonging its lifetime.

Lastly, notice that if no borrowed or ownership types appear in a type, then it is GC. Substituting "GC" with "default-copiable", this is the same design that made the addition of non-copiable classes in C++11 backwards-compatible with C++98.

3 Aside: history of move semantics and resource polymorphism

3.1 The promises of linear logic

It has been suggested from the beginning that the design and implementation of functional programming languages could take inspiration from linear logic (Girard, 1987) and its decomposition of intuitionistic logic. Lafont (1988) took inspiration from intuitionistic linear logic to propose the mixing of strict and lazy evaluation as well as GC-less automatic memory allocation, and justify in-place update of linear values. Safety of parallel and concurrent programs by means of static typing has been another promise of linear logic (Abramsky, 1993). These applications are closely related to continuation-passing-style models (Berdine, O'Hearn, Reddy, and Thielecke, 2000) and syntactic control of interference (Reynolds, 1978; O'Hearn, Power, Takeyama, and Tennent, 1999).

In a series of visionary articles, Baker (1994a,b, 1995) has proposed the integration in functional programming of concepts and implementation techniques from systems programming, using abstractions directly inspired from linear logic and supported by an extensive bibliography of implementation techniques spanning more than three decades. He described many ideas that are now at the basis of the C++11 and Rust resource-management models.

Moving values, Baker (1994a) argues, is a more fundamental operation than copying, and can be implemented by permutations of the stack reminiscent of the structural rules of linear logic. Copying should be explicit, or disabled when meaningless. It is noticed that this linear treatment supports C++-like destructors, and helps avoid synchronisation. It is also noticed that tail-call optimisation, far from being hindered, is in fact the default behaviour in this model.

Baker (1994b) describes what is essentially the modern usage of reference-counted pointers in C++11 and Rust, in which an alternation of moving, borrowing, and deferred copying, is used to minimise reference-count updates. Moreover it is suggested that similar linearity considerations can also be useful for tracing GCs.

In Baker (1995), linear values are advocated as a modular abstraction for resources in the sense of systems programming. Swapping with an empty value is mentioned as an alternative to permuting the stack. Linear abstract data types are proposed as a way to enforce the linearity of types that mix linear and non-linear components by protecting the underlying representation. Linearity of continuations justifies efficient implementations of control operators. The compatibility of the model with exceptions and non-local exits similarly to C++ destructors is mentioned, as well as the added expressiveness of moving resources compared to old C++. And more: only limited accounts of these rich texts can be offered here.

It is clear, at least, that Baker has made the connection between resource management and linear logic, including the compatibility with RAII, and invented move semantics in the process.

Baker admits that the linear discipline is heavy. The only way to pass an argument is to move it, and functions have to return unconsumed arguments alongside the return value in a tuple. Borrowing, which had been considered for reference counting, has not been considered for resources. Minsky (1996) proposed the *unique pointer*, precursor to the C++11 *unique_ptr*, which relaxes the move-only discipline by allowing *non-consumable parameters*, essentially the possibility to pass the resource by copiable reference. In order to ensure the absence of use after free, references to unique pointers are subject to drastic syntactic usage restrictions. Static analyses in the style of Cyclone (Jim et al., 2002) had not emerged yet. Hinnant et al. (2002) managed to integrate move semantics in a backwards-compatible extension of C++ including *unique_ptr*.

These works of Baker were perhaps too in advance of their time. We have found no mention of them in the rest of the literature about applications of linearity in functional programming. Otherwise, when they were mentioned, it was to succinctly point out their limitations.[7] They do not appear to have been accounted for what they are: considerations of language design for resource management, inspired by a connection between foundational works in logic and the practice in PL and systems implementations. Here, too, resource management is seen as more general than control of aliasing, and linear logic inspires aspects of language design that come before the type system.

3.2 Resource polymorphism

Many lessons on substructural type systems are summarised in Walker (2005), such as the practical imperative of a notion of polymorphism for the various linearity restrictions, or the interpretation of the exponential modality of linear logic as reference-counting by Chirimar, Gunter, and Riecke (1996). The latter interpretation suggests an analogy between resource modalities and *smart pointers*, which hinted at a further understanding of practical resource management from the point of view of Girard's polarities.

To sum up, the proposed notion of resource polymorphism is supported by three features:

1. Resource management modes are polarities.
2. Polarity tables define a polymorphism of data types.
3. A notion of subtyping between polarities extends polymorphism: especially, GCed values are both owned and borrowed.

This phrasing uses concepts from Girard (1991, 1993), but these three features are at the basis of C++'s RAII and its later extension with move semantics. For instance, (1.) In C++, default

[7]Minsky, 1996; Clarke et al., 1998; Clarke and Wrigstad, 2003, and other articles remote from the current discussion, mostly in the context of control of aliasing, often in the context of object-oriented programming.

destructors and copy operations are automatically defined, and (2.) in C++11 the non-copiable character of a type is inherited. In addition, (3.) containers such as *std::vector* are polymorphic in the resource management mode: copy operations are disabled at compilation when they are meaningless, using the SFINAE idiom. These features are also at work in Rust (where traits such as *Copy*, *Drop*, or *Sized*, play the role of polarities), and, beyond RAII-based languages, they can be seen to some extent in Clean, ATS[8], and others.

The view of GC types as simultaneously owning and borrowing can be approximated with reference-counting pointers in C++ and Rust, although this usage has the practical issues of reference-counted garbage collection, in addition to being syntactically heavy.

Polarity tables can be considered an automatic and predictable selection of the best resource-management mode for a value. In this sense, resource polymorphism is a way to fill the *static-automatic gap* (Proust, 2016) in the design space of resource management, using abstractions that are compositional.

An originality of C++ and Rust's take on linearity is to emphasize the *how?* instead of the *how many?*, by assigning a computational contents to the copy, move, and drop operations. It is an old folklore in linear logic that distinct exponential modalities can coexist, so any interpretation in terms of "counting the uses" has to miss part of the message. Strikingly, RAII can be seen as arising from shifting attention from *can this value be disposed of?* to *how is this value to be disposed of?*.

In contrast, many investigations into linear type systems interpret linearity as counting uses, starting with Wadler (1990). Among the works that are of close interest to this proposal, this is the case in Kobayashi (1999), Hofmann (2000), Shi and Xi (2013) and Tov and Pucella (2011). Despite this limitation, they all present interesting use cases of linear/affine types, such as capabilities, optimisations and finer memory management. As an exception, there is a qualitative (as opposed to quantitative) interpretation of substructural type systems with type classes in Gan, Tov, and Morrisett (2014), which mentions the analogy with C++'s custom copy and destruction operators, although it misses developed examples, and is not designed for exception-safety. In this proposal, the source of inspiration for paying attention to the qualitative vs. quantitative aspects are constructions of actual models of linear logic where this computational contents appears, such as those of Bierman or Lafont (see Melliès, 2009, for a survey). From this angle, C++ move operators are analogous to monoidal symmetry.

In C++ and Rust, parametric resource-polymorphism (3.) is obtained with templates, with usual limitations and drawbacks (duplication, non scalable to richer type systems, and in the case of C++, poor error messages and slowness at compilation). In a language with proper parametric polymorphism and abstract types, type variables must be given polarities. Cyclone proposes a notion of subkinding (Grossman, 2006), and a similar feature closer to our context was further explored in Alms (Tov and Pucella, 2011). A notion of polymorphism of calling conventions, expressed as a polymorphism of kinds (as opposed to types), was developed in Eisenberg and Peyton Jones (2017), and later reused for *multiplicity polymorphism* in Bernardy, Boespflug, Newton, Peyton Jones, and Spiwack (2018).

The current proposal requires an approach that scales to the qualitative interpretation of linearity; essentially, destructors need to be passed when instantiating ownership type variables.

[8]http://www.ats-lang.org, Zhu and Xi (2005).

Here, functions polymorphic in an ownership type are proposed to depend on an implicit module supplying the destructor. This idea, for which the design in Grossman's and Tov and Pucella's is the most fitting, was explored in Gan et al. (2014) with type classes. In terms of a runtime model, this idea is also similar in essence to the tag-free approach in Morrisett (1995), however applied to destructors only. Tov and Pucella (2011) is also a source of inspiration for showing the existence of principal usage qualifiers with the subkinding approach, and it will come back several times in the rest of this proposal.

4 Ownership types: affine types with destructors

Let us now describe additions to OCaml and give examples of uses. The code given throughout is of two kinds.

```
(* Code aligned to the left: new syntax and examples
   for a language that does not exist yet. *)


              (* Code indented to the right, in italics: a model of
                 the proposed language, given by a whole—program
                 translation in OCaml 4.06. *)
```

The model describes the observational behaviour, but does not respect the runtime model: it is heavy and inefficient. Also it does not check lifetimes, and linearity must be enforced by hand, so it does not model a type system. It models level 2, the language abstractions. It is similar in spirit to Stroustrup's dynamic model of ownership alluded to in Stroustrup et al. (2015): *"useless"*, *"inefficient"*, *"incompatible"*, but *"useful for thinking about ownership"*. As such, it also underlines the limitations of the current OCaml runtime and type system.

4.1 Declaring a custom ownership type

Consider a new type declaration, for affine types given as a pair (A, δ) of a base type A and a user-specified destructor $\delta : A \to \text{unit}$.

type u = **affine**(A, δ)

Let us make more precise what is meant with a pair (A, δ). In Rust's terminology, consider a *Drop* trait.

```
module type Droppable = sig
  type t
  val drop : t —> unit (* must not raise *)
end

type u = affine(M:Droppable)
```

Example: an input file.

```
module Droppable_in_channel : Droppable with type t = in_channel
  = struct
  type t = in_channel
  let drop = close_in_noerr
end

type file_in = affine(Droppable_in_channel)
```

This declaration creates a new type u (= file_in). It has the same runtime representation as M.t (= in_channel). However, being a new type has two consequences: one cannot pass an in_channel to a function that expects a file_in, and one can define two different types of resource with different destructors over the same base type.

We do the same in the dynamic model:

```
module type Droppable = sig
  type t
  val drop : t -> unit (* must not raise *)
end

module Droppable_in_channel : Droppable with type t = in_channel
  = struct
  type t = in_channel
  let drop = close_in_noerr
end
```

To each affine type, associate a module as follows:

```
module type Affine = sig
  type t
  (* ownership type *)
  type t_ref
  (* associated borrow type *)
  val create : t_ref -> t
  (* Create a resource *)
  val borrow : t -> t_ref
  (* Borrow a resource. Raises Use_after_move if the resource has been
     moved or destroyed meanwhile. If the borrowed value is used after
     the resource has been destroyed, this is an error which cannot be
     detected at runtime. In the proposed runtime that allocates memory
     with RAII, this can segfault, but in this model everything is GC'd
     so it will only violate user's invariants (we give an example
     later). *)
  val move : t -> t
  (* Move a resource. Using it on a non−live resource raises
     Use_after_free. Any copy not encapsulated in move can result in
     Double_free. *)
  val delete : t −> unit
  (* Compiler−generated destructor. Only to be called by RAII.scope. *)
end
```

The runtime of the dynamic model is as follows:

```
module RAII : sig
  type 'a ptr
  module Make (M : Droppable) : (Affine with type t_ref = M.t)
  (** Make a new ownership type from a droppable type *)
  val handle : (unit −> 'a) −> 'a
  (** Ensures that destructors are called in order when an exception
     escapes. The bodies of try..withs must be wrapped in a call to
     handle. Failure to do so results in leaks when raising an
     exception. *)
  val scope : (module Affine with type t = 'a) −> 'a −> ('a −> 'b) −> 'b
  (** Simulate a scope for a bound owned variable. Any resource not
     encapsulated in such a scope leaks. *)
end
  = struct
  type 'a ptr = 'a cell ref
  and 'a cell = Live of 'a | Moved | Freed
  type destructor_closure = (unit −> unit) Stack.t
  let dcs : destructor_closure Stack.t = Stack.create ()
  module S = Stack
  let push_closure () =
    S.push (S.create ()) dcs
  let destroy_closure () =
    S.iter (fun f −> f ()) (S.pop dcs)
  let push_destructor (f : unit −> unit) =
    S.push f (S.top dcs)
  let pop_destructor () : unit −> unit =
    S.pop (S.top dcs)
  let handle (f : unit −> 'a) : 'a =
    push_closure ();
    match f () with
    | x −> destroy_closure (); x
    | exception e −> destroy_closure (); raise e
  let _ = push_closure (); at_exit destroy_closure
  module Make (M : Droppable) = struct
    type t = M.t ptr
```

```
      type t_ref = M.t
      let create x = ref (Live x)
      let borrow o = match !o with
        | Live x -> x
        | Moved ->
            failwith "use after move"
            (* This can happen when borrowing after a move. Avoided using:
               a linearity checker. *)
        | Freed ->
            failwith "use after free"
            (* This can happen after copying a resource. Avoided by: the
               language, by always moving. This does not account for all
               use-after-free bugs: the hard ones, which we cannot detect
               at runtime, are obtained when the borrow is created
               during the lifetime of the resource, but used after it is
               destroyed. The latter has to be avoided with a lifetime
               checker. *)
      let move o =
        let o' = create (borrow o) in o := Moved; o'
      let delete o = match !o with
        | Live x -> begin
              o := Freed;
              try M.drop x with
                _ -> ()
                (* An exception is raised while there could already stack
                   unwinding for another raised exception. One can either
                   call it undefined behaviour as in C++ and panic, or
                   specify that all exceptions raised in destructors are
                   silently ignored. *)
          end
        | Moved -> ()
        | Freed ->
            failwith "double free"
            (* This can happen after copying a resource. Avoided by: the
               language, by always moving. *)
    end
    let scope (type a) (module M : Affine with type t = a) x f =
      push_destructor (fun () -> M.delete x);
      let r = f x in
      pop_destructor () ();
      r
  end
```

An affine type is now declared as follows:

```
module File_in = RAII.Make (Droppable_in_channel)
type file_in = File_in.t
let scope_file_in f = RAII.scope (module File_in) f
```

4.2 Creating an owned value

To create a file_in from an in_channel, consider some new syntax:

```
new u(e : M.t)
```

Example: create an opened file and return it as a resource.

```
let open_file name : file_in = new file_in(open_in name)
```

Returning a resource transfers ownership of the resource to the caller.

```
let open_file name : file_in = File_in.create (open_in name)
```

An owned value is destroyed when it goes out of scope without being moved.

```
let drop x = ()
(* all this function does is *not* using the resource x, which is
   already something: it destroys x. *)

let drop2 x y = ()
(* destroy in reverse order of creation (assuming right−to−left
   evaluation): x then y. *)

let fancy_drop x = try (let y = x in raise Exit) with Exit −> ()
(* x is moved into y, and y is dropped by the unwinding mechanism. *)
```

```
let drop x =
  scope_file_in x @@ fun x −>
  ()

let drop2' x y =
  scope_file_in x @@ fun x −>
  scope_file_in y @@ fun y −>
  ()
(* incorrect (for now), can you guess why? *)

let fancy_drop x =
  scope_file_in x @@ fun x −>
```

```
try
  RAII.handle @@ fun () ->
  scope_file_in (File_in.move x) @@ fun y ->
  raise Exit
with
| Exit -> ()
```

4.3 Moving an owned value

An affine value can be moved but not copied.

```
let create_and_move name =
  let f = open_file name in
  drop f

let twice1 name =
  let f = open_file name in
  (f,f) (* typing error: f is affine *)

let twice2 name =
  let f = open_file name in
  drop f;
  f (* typing error: f is affine *)
```

```
let create_and_move name =
  scope_file_in (open_file name) @@ fun f ->
  drop (File_in.move f)

let twice1 name =
  scope_file_in (open_file name) @@ fun f ->
  (File_in.move f, File_in.move f)
(* failure "use after move" *)

let twice2 name =
  scope_file_in (open_file name) @@ fun f ->
  drop (File_in.move f);
  File_in.move f
(* failure "use after move" *)
```

If one copies instead of moving, one can have other kinds of errors. The following two have no source equivalent:

```
let twice3 name =
  scope_file_in (open_file name) @@ fun f −>
  drop f; File_in.move f
(* failure "use after free". *)
let twice4 name =
  scope_file_in (open_file name) @@ fun f −>
  drop f; drop f
(* failure "double free". *)
```

5 Borrowing

A resource cannot be copied, but it can be borrowed. A borrowed value can be copied without restriction, but it cannot be used after its resource is destroyed. A borrowed value does not destroy the resource when it goes out of scope. It is created with the following syntax:

```
&(x : t) : &t
```

The borrow type satisfies

```
&affine(M) = &M.t
```

and, for any GC type t:

```
&t = t
```

At runtime, `&x` and `x` have the same representation. When `x` is owned, the difference lies in `&x` being always copied and `x` being always moved.

5.1 Example: safe reading from a file

Example (compare with the equivalent one in https://ocaml.org/learn/tutorials/file_manipulation.html which has one try/with and two explicit calls to `close_in`):

```
let read_line name =
  let f = open_file name in
  (* if open_file raises an exception, no resource is created. *)
  print_endline (input_line &f);
  (* if input_line raises an exception, f is closed then. *)
  flush stdout
  (* f is closed then. *)
```

```
let read_line name =
  scope_file_in (open_file name) @@ fun f −>
  print_endline (input_line (File_in.borrow f));
```

```
(* if input_line raises an exception, f is closed then *)
flush stdout
(* f is closed then *)
```

5.2 Example: use-after-free

Borrowing can induce more subtle bugs than linearity violations, which requires to check that borrows do not outlive their resource. Cyclone and Rust propose a compositional analysis inspired by type-and-effect systems that assigns lifetime annotations to borrow types.

In OCaml, applying input_line on a closed in_channel gives:

Sys_error "Bad file descriptor"

In contrast, open_file enforces that a file_in is always open (if it is not possible to open it, it safely raises an exception before creating the resource). Thus, this error should not arise.

First, we need to hide the definition of file_in. Indeed, as long as it is known that &file_in = in_channel, then it is possible to let the file handle escape in the old way, given that in_channel is GCed and can therefore be taken possession of freely (it needs to be so, because we are interfacing with the legacy OCaml library: this is just the unsafety of old in_channel surfacing).

```
module File : sig
  type file_in : O
  val open_file : string -> file_in
  val input_line : &file_in -> string
end = struct
  type file_in = affine(Droppable_in_channel)
  let open_file name = new file_in(open_in name)
  let input_line = Pervasives.input_line
end
```

The following program tries to use the resource after it has been freed, and is expected to fail at compilation.

```
let use_after_free name =
  let f = File.open_file name in
  let g = &f in (* g has borrow type &File.file_in *)
  drop f;
  File.input_line g (* lifetime error: g outlives its resource. *)
```

The dynamic model does not perform this static analysis, and is therefore allowed to violate the user's invariant:

```
let use_after_free name =
  scope_file_in (open_file name) @@ fun f ->
  let g = File_in.borrow f in
```

```
drop (File_in.move f);
input_line g (* Exception: Sys_error "Bad file descriptor". *)
```

6 Polarity tables and pattern matching

6.1 Pair of owned

Recall the polarity tables from Section 1. The pair (x,y) is affine as soon as either x or y is affine. Its compiler-generated destructor is obtained by combining those of x and y in reverse order of creation (after testing for zero, and before deallocating the cell).

Implicit modules are not available yet, so the dynamic model uses plain modules with explicit instantiation.

```
module ATensor (P : Affine) (Q : Affine) : Affine
        with type t = P.t * Q.t
        with type t_ref = P.t_ref * Q.t_ref
  = struct
  type t = P.t * Q.t
  type t_ref = P.t_ref * Q.t_ref
  let create (x,y) = (P.create x, Q.create y)
  let borrow (x,y) = (P.borrow x, Q.borrow y)
  let move (x,y) = (P.move x, Q.move y)
  (* What about all the deep copies? In the actual proposed runtime
     model, moves and borrows only involve copying or moving the
     pointer. *)
  let delete (x,y) = P.delete x; Q.delete y
end
```

Example: in the following example, if the first open_file raises an exception, the second one is closed automatically. If the result of this function is later dropped, both files are closed then.

```
let open2 name1 name2 = (open_file name1, open_file name2)
```

```
let open2 name1 name2 =
  let module File2 = ATensor (File_in) (File_in) in
  scope_file_in (open_file name2) @@ fun y ->
  scope_file_in (open_file name1) @@ fun x ->
  File2.move (x,y)
```

6.2 Heterogeneous pair

If the pair is heterogeneous, e.g. x is owned and y is GCed or borrowed, then an implicit coercion of y from GCed to owned is introduced, by registering a root and assigning a destructor that unregisters the root. The runtime representation of the value remains the same. These runtime details do not appear in the model, and the coercion from GCed to affine here is trivial.

```
module Affine_of_GCd (M : sig type t end) : Affine
        with type t = M.t
        with type t_ref = M.t
  = struct
  type t = M.t
  type t_ref = M.t
  let create x = x
  let borrow x = x
  let move x = x
  let delete x = ()
end
```

One can pattern-match on an affine tensor with default destructor, and this involves no additional operation compared to usual pattern-matching. For instance, the following function takes an affine type and returns the string:

```
let fst ((x,y) : string * file_in) : string = x
```

Consistently with the type which indicates that the result is not a resource, the result value can be copied without restrictions.

In the model, this has been expressed by defining Tensor.t and Affine_of_GCd.t non-abstractly.

```
module Affine_of_String =
  Affine_of_GCd (struct type t = string end)
module String_and_file =
  ATensor (Affine_of_GCd (struct type t = string end)) (File_in)
let fst (z : string * file_in) : string =
  RAII.scope (module String_and_file) z @@ fun (x,y) −>
  x
```

When returning the second component instead, the resource y is still alive after the destructor of (x,y) runs: indeed, when that one runs, y has already moved, and in the proposed runtime, all that the destructor sees in place of y is a null pointer, which it ignores.

```
let snd ((x,y) : string * file_in) : file_in = y
```

```
let snd (z : string * file_in) : file_in =
  RAII.scope (module String_and_file) z @@ fun (x,y) −>
  File_in.move y
```

In contrast, one cannot pattern-match on a custom affine value whose underlying type is a tensor, but only on borrows of such values.

6.3 Sums

One can define sum, option and list types similarly:

```
type ('a,'b) sum = Left of 'a | Right of 'b

module ASum (P : Affine) (Q : Affine)
      : Affine
      with type t = (P.t, Q.t) sum
      with type t_ref = (P.t_ref, Q.t_ref) sum
  = struct
  type t = (P.t, Q.t) sum
  type t_ref = (P.t_ref, Q.t_ref) sum
  let create = function
    | Left x -> Left (P.create x)
    | Right y -> Right (Q.create y)
  let borrow = function
    | Left x -> Left (P.borrow x)
    | Right y -> Right (Q.borrow y)
  let move = function
    | Left x -> Left (P.move x)
    | Right y -> Right (Q.move y)
  let delete = function
    | Left x -> P.delete x
    | Right y -> Q.delete y
end

(* more generally *)
module From_map (M : sig
           type 'a t
           val map : ('a -> 'b) -> 'a t -> 'b t
         end) (P : Affine) : Affine
      with type t = P.t M.t
      with type t_ref = P.t_ref M.t
  = struct
  type t = P.t M.t
  type t_ref = P.t_ref M.t
  let create x = M.map P.create x
  let borrow x = M.map P.borrow x
  let move x = M.map P.move x
  let delete x = let _ = M.map P.delete x in ()
```

```
end

module AOption = From_map (struct
                    type 'a t = 'a option
                    let map f = function
                      | Some x -> Some (f x)
                      | None -> None
                  end)

module AList = From_map (struct
                  type 'a t = 'a list
                  let map = List.map (* destroys in LIFO order *)
                end)
```

6.4 Example: a resource-safe interface to Mutex

Using RAII one can ensure that all locks are released. This example could be given earlier, except for try_unlock which returns an affine option type.

First we recall the mutex signature from the library.

```
module type Mutex_sig = sig
  type t
  val create : unit -> t
  val lock : t -> unit
  val try_lock : t -> bool
  val unlock : t -> unit
end
```

RAII implementation:

```
module RAII_Mutex : sig
  (* GC'd type: a mutex is not a resource. Its life is prolonged by the locks
     holding it, so there is nothing to do on destruction. *)
  type t
  (* destroying a lock releases it *)
  type lock : O
  val create : unit -> t
  val lock : t -> lock
  val try_lock : t -> lock option (* affine *)
end
= struct
  type t = Mutex.t
  type lock = affine(struct
                       type t = Mutex.t
```

```
                      let drop = Mutex.unlock
                    end)
  let create = Mutex.create
  let lock m = Mutex.lock m; new lock(m)
  let try_lock m = if M.try_lock m then Some(new lock(m)) else None
end
```

```
module RAII_Mutex(Mutex : Mutex_sig) : sig
  type t
  type lock
  module Lock : Affine with type t = lock
  val create : unit −> t
  val lock : t −> lock
  val try_lock : t −> lock option
end = struct
  type t = Mutex.t
  module Lock = RAII.Make(struct
                    type t = Mutex.t
                    let drop = Mutex.unlock
                  end)
  type lock = Lock.t
  let create = Mutex.create
  let lock m = Mutex.lock m; RAII.create m
  let try_lock m = if Mutex.try_lock m then
                     Some(RAII.create m) (* scope is unnecessary
                                            because the context is
                                            pure *)
                   else None
end
```

6.5 Example: try-locking a list of mutexes and releasing them reliably

(While it can be polymorphic in the type of lockable values, it is not done here for simplicity.)

```
module Locking : sig
  type t : O
  val try_lock : RAII_Mutex.t list −> t option
end
  = struct
  type t = RAII_Mutex.lock list (* by making t abstract, one ensures that the
                                   order does not change. *)
  let try_lock ms =
    let try_with_exn m = match RAII_Mutex.try_lock m with
```

```
      | Some l −> l
      | None −> raise Exit
    in
    try
      Some (List.rev_map try_with_exn ms)
    with
      Exit −> None
      (* all locked mutexes have been released *)
end
```

In particular the locks are released in reverse order. While not important for mutexes, the ability to enforce the order of destruction can be important for some other resources (think of transactions that need to be rolled back).

```
module Locking (Mutex : Mutex_sig) : sig
  type t
  module T : Affine with type t = t
  val try_lock : RAII_Mutex(Mutex).t list −> T.t option
end
  = struct
  module M = RAII_Mutex (Mutex)
  module T = AList (M.Lock)
  module O = AOption (M.Lock)
  type t = M.lock list
  let scope x = RAII.scope (module T) x
  let try_lock ms =
    let try_with_exn (m : M.t) : M.lock =
      RAII.scope (module O) (M.try_lock m) @@ function
       | Some l −> M.Lock.move l
       | None −> raise Exit
    in
    try RAII.handle (fun () −> Some (List.rev_map try_with_exn ms))
    (* Wrong: assumes a ressource−aware List.rev_map, that releases
       partial lockings in case of failure. It is given further below. *)
    with Exit −> None
end
```

6.6 Borrowed values and pattern-matching

When borrowing a pair, one gets a pair of borrows.

```
&(x : a * b) : &a * &b
&(x : string * file_in) : string * &file_in
```

In other words, & is a homomorphism from affine to copiable types. (Remember that &string = string due to its G polarity.) This allows us to pattern-match on borrows.

```
match &(x : a * b) with (y : &a, z : &b) —> (z, y, z) : &b * &a * &b
match &(x : string * file_in) with (y : string, z : &file_in) —> (z, y) : &file_in * string
```

In the above example, the value of type &b * &a * &b, of polarity B, is allocated with the GC (see Table 1 on page 11). Thus, a pair of borrows can either be allocated with RAII, or with the GC: the allocation method of a value of polarity B is not always statically known. In particular,

```
string * &file_in ≠ string * in_channel
```

Indeed, string * in_channel is of polarity G, and always allocated with the GC, whereas string * &file_in can be obtained by borrowing an owned pair. The equation

```
&file_in = in_channel
```

only holds in outermost position in the type, as in:

```
(fun (_ : string, z : &file_in) —> z) : string * &file_in —> in_channel
```

Other data types are treated similarly. The following are implicit definitions in the language.

```
type ('a,'b) &sum = Left of &'a | Right of &'b (* = (&'a,&'b) sum *)
type 'a &option = Some of &'a | None (* = &'a option *)
type 'a &list = [] | (::) of &'a * 'a &list (* = &'a list *)
```

From the point of view of types, this design is likely to raise interesting questions for type inference, with variants to investigate.

6.7 Example: Zipper

As an example of application of this design, it is possible to explore an owned tree with a Zipper (Huet, 1997). Its implementation is not reproduced here because it is identical to the original one, up to checking of lifetimes; we also assume it polymorphic in a sense made more precise in Section 7. Then one can have an owned Zipper that takes ownership of the tree. But one can also take a borrowed Zipper to explore the owned tree. Then the initial Zipper is obtained at no cost by borrowing the owned tree. It is therefore entirely allocated with RAII. Subsequent Zippers are obtained by allocating new values with the GC, and therefore they are allocated in part with the GC and in part with RAII. Thus, the same polymorphic Zipper can be used both with owned and borrowed resources, with in both cases the cost properties expected from a Zipper.

6.8 Other data types

Extending this approach to all OCaml data types raises interesting questions, for instance with abstract types and GADTs. By adopting a design à la Tov and Pucella (2011), in which the polarity can depend on type variables with an operator <'a> ("the polarity of 'a"), it is possible to declare that an abstract type has the polarity of its argument:

```
type 'a t : <'a>
```

or infer that the equality type is GCed even if it is an equality between ownership types:

```
type (_, _) eq (* : G *) = Refl : ('a : O, 'a : O) eq
```

It is also possible to reflect the polarity of phantom types in the type of the GADT with the following idea:

```
type 'a gadt : <'a> =
  | GCd_value : ('b : G) —> unit gadt
  | Owned_value : ('b : O) —> owned_unit gadt
```

where owned_unit is a dummy type of Owned polarity.

```
type owned_unit = affine(struct type t = unit let drop () = () end)
```

It remains to be seen how polarities scale to all OCaml data-types; but in case of stumbling blocks, polarity tables always leave the option of disabling certain combinations.

6.9 Example: capabilities

An example of application of an affine GADT is to encode an existential type, and can be used to tie a capability to a data structure (although this was not the original intent of the proposal). The following example is from Tov and Pucella (2011).

```
module CapArray : sig
  type ('a,'b) t
  type 'b cap : O
  type _ cap_array = Cap_array : ('a,'b) t * 'b cap —> 'a cap_array
  (* inferred as Ownership *)
  val make : int —> 'a —> 'a cap_array
  val set : ('a,'b) t —> int —> 'a —> 'b cap —> 'b cap
  val get : ('a,'b) t —> int —> 'b cap —> 'a * 'b cap
  val dirty_get : ('a,'b) t —> int —> 'a
end = struct
  type ('a,'b) t = 'a array
  type 'a cap = affine(struct type t = unit let drop _ = () end)
  type _ cap_array = Cap_array : ('a,'b) t * 'b cap —> 'a cap_array
```

```
    let make n x = Cap_array (Array.make n x, new cap())
    let set a n x _ = Array.set a n x
    let get a n _ = (Array.get a n, ())
    let dirty_get = Array.get
  end
```

Nothing spectacular happens inside the dynamic model, because capabilities do not require RAII.

```
module CapArray : sig
  type ('a,'b) t
  type 'b cap
  type 'b cap_ref (* One cannot do anything with a &cap, but we
                   need it for defining &cap_array *)
  module Cap (B : sig type b end) : Affine
        with type t = B.b cap
        with type t_ref = B.b cap_ref
  type _ cap_array = Cap_array : ('a,'b) t * 'b cap −> 'a cap_array
  type _ cap_array_ref =
    Cap_array_ref : ('a,'b) t * 'b cap_ref −> 'a cap_array_ref
  module Cap_Array (A : sig type a end) : Affine
        with type t = A.a cap_array
        with type t_ref = A.a cap_array_ref
  val make : int −> 'a −> 'a cap_array
  val set : ('a,'b) t −> int −> 'a −> 'b cap −> 'b cap
  val get : ('a,'b) t −> int −> 'b cap −> 'a * 'b cap
  val dirty_get : ('a,'b) t −> int −> 'a
end = struct
  type ('a,'b) t = 'a array
  type _ cap = unit
  type 'a cap_ref = 'a cap
  module Cap (B : sig type b end) = struct
    type t = unit
    type t_ref = unit
    let create () = ()
    let borrow () = ()
    let move () = ()
    let delete () = () (* a capability is affine and trivially destructible *)
  end
  type _ cap_array = Cap_array : ('a,'b) t * 'b cap −> 'a cap_array
  type _ cap_array_ref =
    Cap_array_ref : ('a,'b) t * 'b cap_ref −> 'a cap_array_ref
  let f : 'a cap −> (module Affine with type t = 'a cap) = fun (type a) x −>
    let module M = Cap (struct type b = a end) in
    (module M : Affine with type t = a cap)
  module Cap_Array (A : sig type a end) = struct
    type t = A.a cap_array
    type t_ref = A.a cap_array_ref
    let create = function Cap_array_ref (x,c) -> Cap_array (x,c)
    let borrow = function Cap_array (x,c) -> Cap_array_ref (x,c)
    let move x = x
    let delete x = ()
  end
  let make n (type a') (x : a') =
    let module M = Cap_Array (struct type a = a' end) in
    M.move (Cap_array (Array.make n x,()))
  let set ar n x () = Array.set ar n x
  let get ar n () = (Array.get ar n, ())
  let dirty_get ar n = Array.get ar n
end
```

The explicit threading in set and get is reminiscent of the shortcoming of ownership without borrowing in Baker (1995). A variation on this idea is to consider affine borrows &mut as in Rust. In Rust, &mut is used for read-write operations whereas & is usually restricted to be read-only. Values cannot be borrowed both with &mut and & at the same time. Together with aliasing control provided by linearity, this prevents data races on resources, and similar issues such as iterator invalidation. There is no obstacle to including &mut in the current proposal. We come back to aliasing control in Section 13.3.

7 Parametric resource polymorphism

Let us go back to the drop example. Make it polymorphic.

```
let drop (type a : O) (x : a) = ()
(* val drop : ('a : O).'a -> unit = <fun> *)
```

In the proposed runtime model, drop already knows how to move or borrow resources of type 'a, because these operations are the same for all types. However, drop needs to know the compiler-generated destructor for 'a. Therefore, the universal quantification on an affine type requires an implicit module supplying the destructor (conversely, abstract types need to supply the destructor).

```
let drop (type a) (module A : Affine with type t = a) (x : a) =
  RAII.scope (module A) x @@ fun x ->
  ()
```

Since any GCed value can trivially be seen as an owned value, drop above is also polymorphic in GCed values and behaves as expected. In this case, a special null value for the destructor indicates at runtime that there is no destructor to run and that allocations have to be done with the GC.

For distinguishing between borrowed and owned variables during type inference, one possibility is that by default inputs are inferred to be borrows, and outputs to be owned. Given that GCed are polymorphically borrowed and owned, this preserves the meaning of current polymorphic code. Then the user can explicitly mark an owned input. Below, the symbol * is used, but we avoid going into details of syntactic choices. (See *"open questions"* for more on this matter.)

7.1 Example: merging two ordered lists

```
let rec merge = function
    | list, []
    | [], list -> list
    | *h1::t1, *h2::t2 ->
        if &h1 <= &h2 then
          h1 :: merge (t1, h2::t2)
        else
          h2 :: merge (h1::t1, t2)
```

```
let merge (type a) (module A : Affine with type t = a) =
  let module AL = AList (A) in
  let module AL2 = ATensor (AL) (AL) in
  let rec iter x =
    RAII.scope (module AL2) x @@ function
     | list, [] | [], list -> AL.move list
     | h1::t1, h2::t2 ->
        if A.borrow h1 <= A.borrow h2 then
          A.move h1 :: iter (AL.move t1, AL.move (h2::t2))
        else
          A.move h2 :: iter (AL.move (h1::t1), AL.move t2)
  in
  iter
```

7.2 Design and implementation

The combination of polarities and parametric polymorphism gives rise to interesting and crucial questions of type inference, principal typing, and polarity specification for abstract types. These questions are addressed by Tov and Pucella (2011) using subkinding and dependent kinds (subtyping of polarities and polarities depending on the polarities of type variables), extending an ML-like design. The hope for the type system (level 1) is that it can be extended without major stumbling blocks.

One question is whether all four polarities $\{G, O, B, O+B\}$ are needed for type variables, or if only two polarities need to be presented to the user (affine/copiable, as in Alms). The difference between 'a:G and 'a:B is that the equation &t=t for t:G can be used during type-checking, e.g. ('a:G) &list = ('a:G) list. It remains to be seen whether this is necessary for expressiveness in concrete situations. But this question might only be superficial: in all cases one wants that the lifetime of a type 'a t : ⟨'a⟩ is deduced from the lifetime of 'a.

As for the efficiency, one will likely want a guarantee that no efficiency is lost due to passing a null destructor to polymorphic functions every time a G is coerced to O. This case can be optimised by treating polymorphic functions on a *"write once, compile twice"* basis: whenever static knowledge allows, one can call a specialised version that does not require the implicit argument, that replaces moves with copies, and that always allocates with the GC. The specialised version corresponds to what OCaml would compile to currently. This optimisation is likely to apply often: it will preserve the efficiency of current OCaml programs, and will also apply in all cases involving borrow types.

8 Owned mutable state

Let us implement a resource-aware Stack module.

8.1 Owned mutable cells

Take OCaml's type of stacks:

```
type 'a t : <'a> = { mutable c : 'a list; mutable len : int; }
```

'a is now allowed to be a resource: polarity tables are the same with or without the mutable keyword.

In addition, if a is owning, then the borrow type a &t is:

```
a &t = { &mutable c : a list; &mutable len : int; }
```

where &mutable is a new keyword. In other words the computation of & stops at the mutable field. Both mutable and &mutable cells are lvalues of the specified type. In addition, when used as an rvalue, the &mutable cell has the borrow type of its contents:

```
(x : 'a &t).c : 'a &list
```

such that (&x).c and &(x.c) are (intuitively) equivalent.

As in Rust and Alms, the primitive operation is swapping between two lvalues:

```
s1.c <—> s2.c
```

Indeed, if the location contains a resource, swapping is more expressive than (<-) because it does not involve any destruction of resource. (s1.c <- l) can be expressed as follows:

```
s1.c <—> (ref l).contents
```

A mutable cell owns its resource, and therefore (<-) proceeds with the destruction of the previous value. (Unfortunately, defining it as a function of <-> is impossible without adding support in OCaml for lvalues as arguments of functions).

8.2 Example: Stack

One idiom in Rust for dealing with mutable structures is to temporarily take ownership of the contents, as with take and swap below. Making use of this idiom in push, pop, and clear is the only substantial difference with the original OCaml Stack module.[9]

In the example below, 'a is considered to be of most generic polarity (O+B), although for backwards-compatibility one will need to reserve 'a for variables of polarity G, and find another notation (''a or 'a : A).

```
module Stack : sig
  type 'a t : <'a>
  val create : unit -> 'a t
  val swap : 'a &t -> 'a &t -> unit
  (** Exchange the contents of two stacks *)
  val take : 'a &t -> 'a t
  (** Take possession of the contents of the given stack.
      The argument is emptied in the process *)
  val push : 'a -> 'a &t -> unit
  val pop : 'a &t -> 'a
  val top : 'a &t -> &'a
  val clear : 'a &t -> unit
  val copy : 'a t -> 'a t
  val is_empty : 'a &t -> bool
  val length : 'a &t -> int
  val iter : (&'a -> unit) -> 'a &t -> unit
  val fold : ('b -> &'a -> 'b) -> 'b -> 'a &t -> 'b
end = struct
  type 'a t = { mutable c : 'a list; mutable len : int; }
  let create () = { c = []; len = 0; }
  let swap s1 s2 =
    s1.c <-> s2.c;
    s1.len <-> s2.len
  let take s =
    let s' = create () in
    swap s &s';
    s'
  let clear s = let _ = take s in ()
  let copy *s = { c = s.c; len = s.len; }
  let push *x s =
    let s' = take s in
    s.c <- x :: s'.c;
    s.len <- s'.len + 1
```

[9]One can also compare it with a similar data structure written in Rust: http://cglab.ca/~abeinges/blah/too-many-lists/book/first-final.html.

```
  let pop s =
    let s' = take s in
    match s'.c with
    | hd::tl —> s.c <— tl; s.len <— s'.len — 1; hd
    | [] —> raise Empty
  let top s =
    match s.c with
    | hd::_ —> hd
    | [] —> raise Empty
  let is_empty s = (s.c = [])
  let length s = s.len
  let iter f s = List.iter f s.c
  (* anticipating a resource—aware iter implementation *)
  let fold f acc s = List.fold_left f acc s.c
  (* anticipating a resource—aware fold_left implementation *)
end
```

In the dynamic model, let us implement a resource-polymorphic ref following this model. It is essentially aref from Alms enriched with RAII support and a swap function that operates on borrowed values.

```
type 'a ref = { mutable contents: 'a }
let ref *x = { contents = x }
(* val ref : ('a : O).'a —> 'a ref = <fun> *)
let (:=) r x = r.contents <— x
(* val ( := ) : ('a : O).'a &ref —> 'a —> unit = <fun> *)
let (!) *r = r.contents
(* val ( ! ) : ('a : O).'a ref —> 'a = <fun> *)
let (!&) r = r.contents
(* val ( !& ) : ('a : O).'a &ref —> &'a = <fun> *)
let swap r r' = r.contents <—> r'.contents
(* val swap : ('a : O).'a &ref —> 'a &ref —> unit = <fun> *)
```

```
module Mutable (P : Affine) : sig
  module T : Affine
  val ref : P.t —> T.t
  val (:=) : T.t_ref —> P.t —> unit
  val (!) : T.t —> P.t
  val (!&) : T.t_ref —> P.t_ref
  val swap : T.t_ref —> T.t_ref —> unit
end = struct
  type t' = { mutable contents: P.t }
  module T = struct
```

```
    type t = t'
    type t_ref = t
    let create r = r
    let borrow r = r
    let move r = { contents = P.move r.contents }
    let delete r = P.delete r.contents
  end
  let ref x = { contents = P.move x }
  let (!) r = P.move r.contents
  let (!&) r = P.borrow r.contents
  let (:=) r x =
    RAII.scope (module P) (!r) @@ fun _ –>
    r.contents <– P.move x
  let swap r r' =
    let x = r.contents in
    r.contents <– r'.contents;
    r'.contents <– x
end
```

8.3 Unsafe

For a performance-critical library such as Stack, it is preferable to have an unsafe...end block à la Rust, and write the more efficient:

```
let push *x s = unsafe s.c <– x :: s.c end; s.len <– s.len + 1
```

In an unsafe block, the checking of lifetimes and ownership is reverted back to the user. The programmer can (and is encouraged to) reason about the correctness of their code by showing it equivalent to the previous one that uses take. In addition to being more efficient, this code coincides on GCed values with the one in OCaml's Stack implementation. unsafe...end can be considered without altering the rest of the resource-management model.

9 Closures

Recall drop2:

```
let drop2 x y = ()
```

When will the following partial application destroy x?

```
let f = drop2 x in ...
```

when drop2 is applied to x, or when the scope of f ends? One criterion is that Currification preserves the meaning. Then, partial application must have the meaning of:

```
let drop2 x = fun y -> let _ = x in ()
```

and therefore drop x only once y has been passed.

9.1 Owning closures

As in C++/Rust, closures can take possession of their variables. A function such as the above where the captured variable is not used is not necessarily a fabricated corner case: in RAII it is common to use guards, that is values whose sole purpose is to exist for the duration of a scope and not be used otherwise.

These languages provide a syntax to decide whether a variable has to be moved, copied, or borrowed in a closure. This is always expressible with lets (as illustrated above), so the problem of expressing which affine variables are captured is reduced to a question of syntactic sugar (into which this proposal does not go).

A consequence of the closure owning their resources is that function values have a polarity determined by the closure (and neither by their argument nor by their return type). This is predicted by the linear call-by-push-value model. In semantic terms, a linear call-by-value function has polarised type:

$$\Downarrow (X \multimap Y)$$

(readers of Levy might write its non-linear version $G(X \to FY)$, the F is implicit because it can be deduced from the context).

Here $\Downarrow$ is the type constructor of closures. In linear call-by-push-value, there are several types of closures: linear ($\Downarrow$), copiable (!). The current OCaml function type, copiable at will, is:

$$X \to Y = !(X \multimap Y)$$

This leads to the introduction of an affine function type, as in Alms, or as FnOnce in Rust. Affine means that the function has to be used at most once (but it can use its argument as many times as the argument's polarity allows).

$$X \to_1 Y = \Downarrow(X \multimap Y)$$

Thus drop2 can be given type:

```
drop2 : ('a : O) ('b : O). 'a -> 'b ->₁ unit
```

denoting that the second closure holds a resource which will be consumed, in this case the first argument.

Now what if an affine function value only uses its closure by borrowing? In that case, although the closure is affine, it makes sense to call the function several times. A third function space $\to_A$ is introduced, and is similar to $\to_1$ except that borrowed values of type &(a $\to_A$ b) can be applied to values of type a as well. This is Fn in Rust. (Let us hope this is enough function spaces!)

```
module AFun : sig
  type ('a,'b) fn
  type ('a,'b) fn_ref
  type ('a,'b) fn_once
  module Fn (M : sig type a type b end) : Affine
          with type t = (M.a, M.b) fn
          with type t_ref = (M.a, M.b) fn_ref
  module FnOnce (M : sig type a type b end) : Affine
          with type t = (M.a, M.b) fn_once
  val make : ('a −> 'b) −> ('a,'b) fn
  val make_once : ('a −> 'b) −> ('a,'b) fn_once
  val move_into_once : (module Affine with type t = 'c) −>
                       'c −> ('c −> ('a,'b) fn_once) −> ('a,'b) fn_once
  val move_into : (module Affine with type t = 'c
                                 and type t_ref = 'd) −>
                  'c −> ('d −> ('a,'b) fn) −> ('a,'b) fn
  val app_ref : ('a,'b) fn_ref −> 'a −> 'b
  val app_once : ('a,'b) fn_once −> 'a −> 'b
end = struct
  module type Closure = sig
    module A : Affine
    val x : A.t
  end
  let closure_move (module M : Closure) = (module struct
                                             module A = M.A
                                             let x = A.move M.x
                                           end : Closure)
  let closure_delete (module M : Closure) = M.(A.delete x)
  type ('a,'b) fn = ('a −> 'b) * (module Closure) list
  type ('a,'b) fn_ref = 'a −> 'b
  type ('a,'b) fn_once = ('a,'b) fn
  module Fn (M : sig type a type b end) : Affine
          with type t = (M.a, M.b) fn
          with type t_ref = (M.a, M.b) fn_ref
    = struct
    type t = (M.a, M.b) fn
    type t_ref = M.a −> M.b
    let borrow (f,l) = f
    let move (f,l) = (f, List.map closure_move l)
    let delete (f,l) = List.iter closure_delete l
  end
  module FnOnce (M : sig type a type b end) : Affine
          with type t = (M.a, M.b) fn_once
    = Fn (M)
  let make f = (f,[])
  let make_once f = (f,[])
  let move_into_once (type c) (module A : Affine with type t = c)
        res closure =
    let x = A.move res in
    let (f,l) = closure x in
    let x_mod = (module struct
                   module A = A
                   let x = x
                 end : Closure)
    in
    (f,x_mod::l)
  let move_into (type c) (type d)
        (module A : Affine with type t = c and type t_ref = d)
        res closure =
    let x = A.move res in
    let (f,l) = closure (A.borrow x) in
    let x_mod = (module struct
                   module A = A
                   let x = x
                 end : Closure)
    in
    (f,x_mod::l)
  let app_ref = (@@)
  let app_once (type a') (type b') fn x =
    let module F = Fn (struct type a = a' type b = b' end) in
    RAII.scope (module F) fn @@ fun (f,_) ->
    f x
end

let drop2 x =
  scope_file_in x @@ fun x ->
  AFun.move_into_once (module File_in) x (fun x ->
      AFun.make_once (fun y ->
          (scope_file_in y @@ fun y -> ())))
(* : File_in.t -> (File_in.t, unit) AFun.fn_once *)
```

9.2 Functions with static closures

What is the meaning of LCBPV's negative ⊸ then? Interpret it as the type of functions not yet wrapped into (dynamic) closures, as a distinction reminiscent of the one between function pointers and closures seen in C++ and Rust.

The introduction of the closure can be statically delayed until a value is actually required for the function, that is, when the function is passed to another function, wrapped into a data structure, or fully applied. (A full application is one which results in an expression with positive type.)

At either of these points, the set of free variables used in the expression is known statically, and therefore introducing the closure can be done then.

There are at least three reasons for introducing the negative $\multimap$ function space. If the function uses a resource, introducing the closure forces to move the resource. Introducing it early can move the resource earlier than necessary. For instance:

```
let f *x =
  let g y = &x in
  h &x (* is x still live or has it moved into the closure for g already? *)
  g
```

Delaying the introduction of the closure until the last line lets us accept this program.

The second reason is that by typing differently returned functions that do not need a closure yet, one reduces the number of intermediate closures in call-by-value, statically and compositionally, unlike the (non-compositional) nested redex optimisation (Danvy and Nielsen, 2005) or in the (dynamic) ZINC machine (Leroy, 1990).

This idea sheds light on the right-to-left evaluation order of arguments seen in the ZINC. Indeed, it naturally leads to a right-to-left order, because in the double application:

```
f e1 e2
```

the type of (f e1) is negative and requires waiting for the value of e2, and because left-to-right evaluation:

```
let x1 = e1 in ... let xn = en in f x1 ... xn
```

is not macro-expressible given that its size is linear in the number of arguments. (The idea of relating optimisations of closure introduction in call-by-value to call-by-push-value is not new, a sketch of a relationship with the nested redex optimisation and the ZINC machine is given in Spiwack, 2014, and a different computational description for the CBPV arrow is sketched.)

The third reason is an extension of the second one, in the presence of affine closures. Tov and Pucella (2011) have noticed that currified functions tend to accumulate annotations, often in a predictable manner, e.g.:

$$\forall\alpha\beta.(\alpha \to \beta \to_{\langle\alpha\rangle} t \to_{\langle\alpha\rangle+\langle\beta\rangle} u)$$

where $\to_{\langle\alpha\rangle}$ denotes the type of closures with the same polarity as α. This makes dependent kinds necessary in their approach. This happens because a function obtained by currying just adds the variable to the closure, as opposed to a function that performs some computation before returning a closure. By introducing the closure in a delayed manner, such a currified function can be typed without annotations:

$$\forall\alpha\beta.(\alpha \multimap \beta \multimap t \multimap u)$$

Then, the closure is created, its variables moved, and its polarity determined, either after full application, or when one tries to pass or return the result of a partial application.

10 Tail-call optimisation and control operators

Tail-call optimisation (TCO) suffers from a bad interaction with destructors. Given that a resource must be destroyed at the end of a scope, it has to remain on the stack and prevents any TCO.

There are three separate issues:

1. This can lead to surprises, when writing polymorphic code or during refactoring. Hence, the user should be allowed to specify that a call is expected to be tail, and get an error if this is not the case.
2. This can prevent desired tail calls. The solution is to have a convenient way to express that resources must be destroyed before the call, rather than after.
3. The last condition being met, one must be able to actually implement TCO.

1) and 2) are largely a matter of syntactic choices and 1) is implemented in OCaml with the [@tailcall] attribute.

10.1 Example: List.rev_map

Below is a solution for 2) and 3) involving an operator tail_call that destroys remaining resources before doing a tail call. In the dynamic model it is implemented with a combination of an exception and usual TCO.

```
let rev_map (type a : O) (type b : O) (f : a −> b) l =
  let rec rmap_f accu = function
    | [] −> accu
    | *a::*l −> tail_call rmap_f (f a :: accu) l
  in
  rmap_f [] l
```

```
let rev_map (type a) (module A : Affine with type t = a)
      (type b) (module B : Affine with type t = b)
      (f : a −> b) l =
  let module AL = AList(A) in
  let module BL = AList(B) in
  let exception Tail_rec of BL.t * AL.t in
  let rmap_f accu x =
    RAII.scope (module BL) accu @@ fun accu −>
    RAII.scope (module AL) x @@ function
     | [] −> BL.move(accu)
     | a::l −> raise (Tail_rec (f (A.move a) :: BL.move(accu), AL.move l))
  in
  let rec tail_rec x y =
    try
      RAII.handle (fun () -> rmap_f x y)
    with
      Tail_rec (x',y') -> (tail_rec [@tailcall]) x' y'
  in
  tail_rec [] l
```

```
let test_rev_map () =
  let x = open_file "/tmp/dummy1" in
  let y = open_file "/tmp/dummy2" in
  let z = open_file "/tmp/dummy3" in
  let rev_files = rev_map (fun *x -> x)
  let l = rev_files (rev_files [x;y;z]) in
  (* print the first lines in order and close the files *)
  rev_map (fun *x -> &x |> input_line |> print_endline) l
```

```
let test_rev_map () =
  let x = open_file "/tmp/dummy1" in
  let y = open_file "/tmp/dummy2" in
  let z = open_file "/tmp/dummy3" in
  let z' = File_in.borrow z in
  let rev_files = rev_map (module File_in) (module File_in)
                    (fun x -> File_in.move x)
  in
  let module L = AList(File_in) in
  let scope = RAII.scope (module L) in
  let _ =
      scope [File_in.move x; File_in.move y; File_in.move z] @@ fun l ->
      scope (rev_files (L.move l)) @@ fun l' ->
      scope (rev_files (L.move l')) @@ fun l'' ->
      (* val l'' : File_in.t list = [<abstr>; <abstr>; <abstr>] *)
      rev_map (module File_in) (module Affine_of_GCd(struct type t = unit end))
        (fun x -> scope_file_in x @@
          fun x -> (x |> File_in.borrow |> input_line |> print_endline))
        (L.move l'')
  in
  (* check that files are closed *)
  print_endline (input_line z')
  (* Exception: Sys_error "Bad file descriptor". (expected) *)
```

10.2 Affine control

More generally, such a tail_call operator (and other operators such as return) can be expressed with a control operator, i.e. a variant of Landin's J operator (Landin, 1965). Griffin (1990) has shown that Scheme's call-with-current-continuation (call/cc) can be typed with:

$$((\alpha\ \mathsf{cont}) \to \alpha) \to \alpha$$

One can give cont a polarity $O + B$ to prevent it from being called several times, passed inside its own arguments, and passed inside the returned value. This ensures that it can be implemented efficiently with stack unwinding, similarly to Scheme's call-with-escape-continuation (call/ec). Such a linear call/cc would provide an efficient statically-typed error-handling mechanism, alternative to exceptions.

One can also give cont the polarity O in the following:

$$((\alpha\ \mathsf{cont}) \to \bot) \to \alpha$$

with the requirement that the control operator starts a new stack (fiber). This linear variant of Felleisen's C operator could be more of use in the context of concurrency.

Thus we conjecture that with appropriate typing, such operators can be implemented at runtime like exceptions or effect handlers, but that a difference appears with typing, as exemplified in the implementation of tail_rec above, which has to rely on a local exception declaration specific to the type. Furthermore, in the presence of negative function spaces (9.2), the type cont can be refined to provide higher-order access to the stack:

$$(\alpha \multimap \beta)\ \mathsf{cont} = \alpha \otimes (\beta\ \mathsf{cont})$$

(Munch-Maccagnoni, 2014). This allows expressing stack inspection, with possible applications to debugging and security (Clements and Felleisen, 2004; Clements, 2006).

Once a type system with lifetimes and linearity is in place, investigating and implementing such control operators is a low-hanging fruit.

11 Example: a reference-counted cache

Reference-counting pointers such as *shared_ptr/weak_ptr* in C++11 and Rc/Arc in Rust are useful for sharing resources compositionally, such as between threads. The resource is freed as soon as the reference count reaches zero. Unlike tracing GCs, they are not restricted to contain non-resource values. For instance, one can easily implement a cache of resources in C++11 as a vector of weak pointers. For non-resources, such a cache can already be implemented in OCaml using GCed weak pointers.

In C++11 and Rust, reference-counting pointers are implemented using custom copy operators that increase the reference count. Customizing copy operations was not explored for several reasons. This requires to enrich the proposal with new polarities, which makes the language more complex, and is likely to have a performance impact (in contrast with languages that implement polymorphism with templates). Also, it is accepted that custom copy and move operations are more error-prone, and are recommended to be seldom used, only for defining new resource modalities when the ones from the libraries do not fit (this is C++11's *rule of zero*, an idiom part of coding guidelines and consolidated in the C++14 standard). For now, omitting customizable copy and move operators still allows explicit copy operations, which can be streamlined with implicit modules, and does not preclude future extensions with new polarities if necessary, which would again be backwards-compatible.

Still, elementary structures for dealing with resources, such as caches, have to be implementable in ways fully integrated with the language. For instance, assume that one is given bindings to Qt's QFileSystemWatcher[10], a platform-independent front-end to file monitoring systems such as Linux's Inotify.

```
module QFileSystemWatcher : sig
  val addPath : string —> bool
  val removePath : string —> bool
  (...)
```

In order to monitor a file, one needs to add its path, and register a callback (using a mechanism not shown above). Monitoring a file has a cost (there is a limit to their number on some platforms) so one wants to end the monitoring as soon as it is no longer needed. This interface is an example of implicit resource-unsafety: indeed, if two parts of a program want to monitor the same file at overlapping moments, then the first one to call removePath will stop the second one from being subsequently notified.

A solution is to define *file monitor handles*, which uniquely handle the presence of a file in the monitoring list, and *file monitor guards*, responsible for handling a certain callback, where each handle keeps track of the number of guards for this file. Therefore, not only we need a cache of file monitor handles, but it has to be flexible enough to allow the definition of custom guards, that are also responsible for de-registering a callback.

As an example, we describe the implementation of a reference-counted cache for resources that does not require customizing copying. Guards are used for establishing the reference count. Let us assume that OCaml's Map module has been extended to contain owned values.

```
module Rc_Cache (Ord : Map.OrderedType) : sig
  type key = Ord.t
  (** The type of cache keys. *)
  type ('a : O) t : O
  (** The type of caches. *)
  type ('a : O) cached : O+B
  (** The type of guards for accessing a cached resource. Holding a guard
      prolongs the life of the resource. The lifetime of the guard is
      limited by that of its cache. *)
  val create : unit —> 'a t
  (** Create an empty cache *)
  val get_or_add : key —> (key —>₁ 'a) —> 'a &t —> 'a cached
```

[10] https://doc.qt.io/qt-5/qfilesystemwatcher.html

```
  (** [get_or_add k f c] retrieves the current cached resource with key [k],
      or associates [k] to [f k] if it does not exist yet, and returns
      a guard to the cached resource. The resource created by [f k] is
      destroyed as soon as there are no more guards pointing to it. *)
  val get : key -> 'a &t -> ('a cached) option
  (** [get k c] retrieves the current cached resource with key [k] if it
      exists. *)
  val deref : 'a &cached -> &'a
  (** Borrow the cached resource. Its lifetime is limited by that
      of its guard. *)
  val copy : 'a &cached -> 'a cached
  (** Get a new guard from an extant guard, increasing the reference count. *)
end = struct
  type key = Ord.t

  module M = Map.Make (Ord)

  type ('a : O) t = ('a * int ref) M.t ref

  let create () = ref (M.empty ())

  let take cache =
    let *cache' = create() in
    cache <-> &cache';
    !cache'

  let decr k cache =
    match M.find k (!& cache) with
    | (_, r) -> begin
        decr r;
        if !r = 0 then
          cache := take cache |> M.remove k
      end
    | exception Not_found -> assert false

  type 'a cached = affine(struct
                           type t = &'a * key * &'a t
                           let drop (_, k, cache) = decr k cache
                         end)

  let deref (x, _, _) = x

  let get k cache =
    match M.find k (!& cache) with
    | (x, r) -> begin
        incr r;
        Some (new cached(x, k, &cache))
      end
    | exception Not_found -> None

  let get_unsafe k cache =
    match get k cache with
    | Some *cached -> cached
    | None -> assert false

  let get_or_add k f cache =
    match get k cache with
    | Some *cached -> cached
    | None -> begin
        cache := take cache |> M.add k (f k, ref 0);
        get_unsafe k cache
      end

  let copy (_, k, cache) =
    get_unsafe k cache
end
```

Remarks:

- To come back to our example with QFileSystemWatcher, guards be customised to perform additional duties such as registering and unregistering callbacks, by composition of ownership types.
- No effort has been made for thread-safety. However, thanks to Map being a persistent data structure, the cache can be made thread-safe by replacing the two ref types in the definition of Rc_Cache.t with atomic references as proposed in ocaml-multicore, and implement synchronisation in take. Further efforts are of course necessary for enabling thread-safe mutation of the contents.
- If separating & into a read-only & and a linear &mut, the compiler would worry about a possible iterator invalidation in decr and in get_or_add. One therefore needs to have extra unsafe...end around the assignments of cache. The soundness of this module has to be established by external means, in the spirit of Benton, Hofmann, and Nigam (2016); Jung, Jourdan, Krebbers, and Dreyer (2018). This is expected, because it implements a resource-management policy which is finer-grained than generic considerations on linearity and lifetimes.

12 Additional considerations

12.1 Interfacing with foreign runtimes

Interfacing with foreign runtimes is made easier with RAII, even those that are GCed, as already demonstrated by Rust. Foreign functions can receive owned values together with their destructors (for instance responsible for unregistering GC roots). This is independent from whether the foreign runtime possesses abstractions for ownership: ownership comes with responsibility.

Conversely, OCaml functions can receive ownership of foreign values. In that case, the destructor generated by the interface (for instance deallocation) is automatically called using RAII.

One benefit of managing the allocation of owned values with RAII is that it could become possible to structurally map types and values when exchanging between the two languages, for the whole phase of the structure which is RAII-allocated, provided the language on the receiving end is rich enough to express the memory layout of the other language.

It would also be interesting to see if by combining the RAII mechanisms of two languages with exceptions that support it, one can achieve interoperability for exceptions independently of their implementation. (For instance for interoperating C++ and OCaml.)

12.2 Unboxing

In OCaml, data structures of size 1 can be unboxed. This does interfere with custom affine types: indeed, checking whether a resource has moved involves checking if the pointer has been set to zero. Therefore, custom affine types have to be taken on boxed types.

Unboxing is the subject of current discussions in OCaml; interest has notably been shown for generalizing unboxing and making it predictable. Seeing things from the angle of polarities suggests a type-directed notion of unboxing fully compatible with the current proposal (in the style of Eisenberg and Peyton Jones, 2017) by considering a polarity of statically-known and fixed memory size (like Rust's *Sized* trait). But there are obstacles, such as the cost of passing the size at runtime to polymorphic functions, and the necessity of occasionally having explicit boxing modalities in the types (so as to avoid the cost of repeatedly boxing and unboxing at function call boundaries).

Unless it is determined that the benefits outweigh the costs, the proof-theoretic angle stops short of providing a solution to boxing-related performance issues in OCaml. Still, it suggests that the proposal might adapt to any sensible design for unboxing.

12.3 Efficient implementation of exception-handling

Let us describe in broad strokes how the implementation of exception-handling can be extended conservatively to support RAII, as illustrated by the dynamic model. With each trap one keeps a map associating registers or stack addresses to their destructors, called above the destructor closure. It is maintained by pushing and popping it according to the lexical scoping of owned variables. Notably it does not need to be updated when a resource is moved, because the destructor already knows this at runtime by testing for null pointers. For this reason the destructor closure can simply be implemented as a stack, that does not need to be accessed during non-exceptional execution, only pushed and popped. Stack unwinding when an exception is being raised, on the other hand, now also involves calling the destructors in orderly fashion, before handling the exception.

The runtime overhead compared to the current implementation is proportional to the number of scopes for owned variables: in particular there is no overhead for current resourceless code. This is close to optimal, given that a destructor runs at the end of every such scope in any case. The same idea can apply in multicore with fibers. In essence, each fiber owns a set of resources, knows which ones it owns, how to destroy them, and in which order. With effect handlers, the destructors run when the captured continuation is dropped.

An alternative implementation model is inspired by permutation stacks (Baker, 1994a). In this model, a stack of affine values would be maintained separately from the stack of copiable values. Moving is done by permuting elements in the stack rather than copying+nullifying, which avoids keeping nulls for values no longer there. This is more elegant because it replaces the destructor closure, but the cost of swapping must be assessed. Again, the costs are only paid by resource-intensive users.

12.4 Test for liveness

A naive affine type system based on affine logic will reject programs similar to the following one even if b() is constant:

```
if b() then f(x) (* moving x *);
if not b() then g(x) (* error: liveness of x is unknown *)
```

Since it is known at runtime whether a resource has moved, expressiveness can greatly be increased by adding a test for liveness that converts the run-time information into a static information.

```
if b() then f(x) (* moving x *);
if live(x) && not b() then g(x) (* moving x *)
```

Situations where the contents of x may have moved, rather than x itself, still have to be prevented statically.

12.5 Assessing the use for RAII allocation

In an alternative representation, everything can be allocated with the GC, and destructors do not manage allocation. However, this change impacts two use cases: 1) Foreign pointers can still be received as resources, and OCaml values can still be passed to a FFI by registering the root; however this prevents any form of structural mapping between OCaml structures and foreign structures; 2) The purpose of treating very large data structures as a resource is to reduce the GC load and perform real-time collection. Moreover, mutation avoids the write barrier of the GC and is therefore more efficient.

RAII memory allocation (as opposed to RAII for non-memory resources) is a low-hanging fruit once destructors are in place, and it is likely useful for specific situations, but its practical impact will need to be assessed concretely.

13 Comparison with existing PLs in a nutshell

While some functional PLs have support for timely and exception-safe release of resources (F#[11], Scala[12]) and there are some idioms for this in OCaml, these languages lack move semantics and resource polymorphism, which gives resources their first-class citizenship in C++11 and Rust. A natural question which we do not explore due to lack of familiarity is whether the current proposal could realistically be applied to them. Let us now focus on comparisons with designs that are state-of-art in this respect.

13.1 Systems PLs (Cyclone, C++11, Rust)

This proposal extends the OCaml philosophy and runtime model. The aim is to improve the resource management of a general-purpose PL, rather than creating a new language dedicated to systems programming. For this reason, compared to systems PLs, it can do with a coarser-grained management of memory representation, and propose GCed allocation for non-resources as the default. In contrast, the semantic perspective suggested that the resource-management mechanisms in C++11 and Rust (RAII and move semantics) could well belong to a high-level, general-purpose programming language.

The language the closest to the proposal in this domain is Rust. Leaving aside the differences in purpose, the novelty compared to Rust is to propose a tight integration of RAII with a tracing GC, both at the level of types (meaning), and at the level of the runtime.

Could the same design be used as a basis for integrating a GC in C++ or Rust? The proposed design is built around the assumption of a tracing GC based on a uniform representation of values, with a run-time tag distinguishing traced pointers from non-traced data. An alternative approach in Rust, proposed by Goregaokar and Layzell with a prototype GC design and implementation[13], considers a *Trace* trait that provides run-time information about the memory layout for those types that are GCed, reminiscent of the tag-free approach (Morrisett, 1995). It is similar in several aspects: reference-counting is used at the interface between the linear and the GCed values to track roots, the use of traits provides some degree of resource polymorphism, and borrowing is proposed as a mean to avoid incrementing reference counts. The syntactic footprint of using Gc pointers is similar to that of reference-counted (Rc) pointers. One way to see it is as an improvement of the latter for those types with trivial destructors, fixing cascades of reference-count updates, latency, and leaks due to cycles. It remains to be seen whether this approach can be made efficient.

In contrast, our proposal integrates with a GC that has already been designed for efficiency. The tagged representation is helpful in avoiding reference count updates when assuming ownership of a GCed value (not just borrowing), stack scanning reduces the need for reference-counted roots, and pointers that move during minor collection can be handled without any added complexity.

[11]http://fsharp.org/. This is not the *destructors* which are finalizers, but the IDisposable interface together with the *use* binder. The back-end appears to support moving (*use* will check for null before destruction), but this does not seem to be reflected in the language.

[12]http://www.scala-lang.org/. This is implemented as a library (ARM). Monadic sequencing seems to be used to ensure the absence of uses after free.

[13]https://github.com/Manishearth/rust-gc/.

Moreover, greater integration of the GC within the language, notably a uniform representation between traced and untraced values, affords a broader notion of polymorphism, such as between GC and borrow types with GC-allocated borrowed values, and resource-polymorphic algorithms and data structures. This is where we believe another potential sweet spot lies for OCaml, with no equivalent among existing PLs.

13.2 Functional PLs with generic linear types (Linear Lisp, Clean, Alms)

Linear or affine types have for a long time been associated with the idea of counting uses, even though foundational investigations on linear logic have from the beginning let us hope of richer interpretations. Baker (1994a,b, 1995), however, saw in them the opportunity to improve on more than three decades of PL and systems implementation techniques (Section 3). Yet the resulting language Linear Lisp did not survive, and the lessons remained misunderstood or ignored. ATS is another language that explores linearity for memory allocation.

As long as their role was merely to decorate programs by counting uses, the applications of linear types remained limited. This works well for linear in-place update, which is explored by Clean and ATS for purity and efficiency. But another prediction of linear logic (not exclusive to it) is well-accepted but not always presented as a form of linear typing: the importance of strict types. In fact, in the linear call-by-push-value model, linear types are, by reminiscence of focusing and polarisation in linear logic, a refinement of strict types (i.e. linear types are positive), and strict types in the style of ML are linear types that are copyable and discardable.

In the area of linear type systems, the closest language to suit the requirements of this proposal is Tov and Pucella's Alms, which to us appears state-of-art in many type-system aspects. Compared to Alms, this proposal adopts a different perspective about affine types. Specifically, this proposal essentially extends Alms by attaching destructors to affine types. The semantic point of view suggests that affine types with destructors are the natural guise of linear types in the presence of control, instead of affine types being at odds with linear types. From this perspective, further connections are drawn between some of the language features and idioms of C++11, and Girard's polarisation. Using these connections, interesting design and applications of affine types for OCaml can be extracted from Stroustrup's RAII and Hinnant et al.'s move semantics. Specifically, not only one retains the uses of Alms, but user-provided and compiler-generated destructors unlock many interesting uses beyond counting, such as custom resources, low-level resources, and RAII allocation for performance and interoperability. This also lets us consider a borrowing mechanism, which in Alms would be vacuous (at least in their copiable form).

This tends to validate Baker's prediction that linear types open up a new dimension for functional PLs. They address actual reported shortcomings of OCaml, with perhaps many uses remaining to be discovered.

Aside: Linear Haskell

Linear Haskell (Bernardy et al., 2018) seems to be an exception to the necessity of strictness among linear type systems, with applications such as pure in-place update. Linear Haskell is *"designed around linear types based on linear logic"*, a notion that the authors oppose to linearity *à la* Alms (§6.1: *"this does not match linear logic (there is no such thing as a linear* *proposition)”*), and to linearity *à la* Rust (§6.3: *“we could have retrofitted uniqueness types to Haskell. But several points guided our choice of designing Linear Haskell around linear logic instead”*). It proposes a linear typing for arguments of functions, rather than linear types per se, in a way reminiscent of the negative (call-by-name) interpretation in linear polarised models (e.g. Melliès and Tabareau, 2010). Linearity is interpreted as counting uses.

One of the main claims in Linear Haskell is a backwards compatibility on the same level of ambition as the C++11 move semantics proposals. However, exceptions in Haskell are not addressed nor mentioned, despite the fact that linearity as opposed to affineness is important for resource-management protocols (§2.3, §5.2), one of the two proposed applications. In addition, a prospective use case is to handle values from foreign runtimes, including GCed ones like in Rust (§7.3). How it would avoid being impacted negatively by exceptions is not detailed.

In light of this proposal, it is natural to ask whether exception-safety can be implemented with destructors. Unfortunately, the meaning of destructors is unclear in the absence of strict types to attach them to: in a lazy language, the notions of lifetimes and scopes are not as clearly related. Laziness both prevents the timely release of owned resources, and tends to let borrowed resources escape, unless special measures are taken to impose sequencing. Furthermore, Linear Haskell proposes a notion of resource polymorphism (multiplicity polymorphism) which describes the mixing of linear and unrestricted arguments to functions for a single resource modality. This is a unique design, for which it is unclear whether it can be adapted for non-idempotent resource modalities, and multiple resource modalities with distinct computational behaviour, as required for this proposal.

Another prospective application of linear typing in Haskell is to address an issue with unpredictable performances (§7.1). At this point it is natural to ask whether it is possible to further take inspiration from linear logic and retrofit Haskell with strict types: they are known to produce more predictable performances, and affine types with destructors would fit naturally. In theory nothing prevents applications of linear types in Haskell as described in this proposal; in practice it likely requires a bigger evolution of the language and libraries. According to Bolingbroke and Peyton Jones (2009): *“We would like to expose the ability to use ‘strict’ types to the compiler user* [...] *but allowing strictness annotations to appear in arbitrary positions in types appears to require ad-hoc polymorphism, and it is not obvious how to go about exposing the extra generality in the source language in a systematic way”*. Starting from a language that already features strict types is an essential aspect making the current proposal practical and realistic.

13.3 Languages with control of aliasing (Vault, Mezzo, Rust)

The notion of linearity developed here can be combined with type abstraction to provide a syntactic control of interference (Reynolds, 1978; O’Hearn et al., 1999), reminiscent of linear abstract data types (Baker, 1995) and external uniqueness (Clarke and Wrigstad, 2003). In this scenario, the corresponding borrow type cannot appear in the interface: there can be as many borrows as one wants, as long as they cannot be used they cannot interfere! This is exemplified with reader/writer locks with capabilities in the style of Alms (Section 6.9).

The Rust language distinguishes linear borrows (`&mut`) from read-only borrows (`&`), a design which can be used to lift the previous restriction on borrows when controlling aliasing. This design has been shown to be a practical and scaleable approach to solving issues such as data races. The latter issue will become all the more important with ocaml-multicore, which clarifies the semantics of data races and provides atomic references (Dolan, Sivaramakrishnan, and Madhavapeddy, 2018), but has to exhort programmers to *"still strive to avoid races"* using locks and atomics[14].

First, there is no obstacle to applying the Rust approach for preventing data races on owned values. As for shared mutable state, OCaml already tends to reduce its need compared to C++11 and Rust, thanks to an efficient support of persistent data structures with sharing. Lastly, although racy by itself, keeping GCed shared local state in the language leaves the door open to its encapsulation in safe abstractions in libraries. The design space remains to be explored: this has not been the goal of this proposal so far. However, the combination of the ingredients we already have at hand promises at least an efficient model for data-race prevention in the style of Rust, without the need for compromising backwards-compatibility of the language. (Libraries, though, might require some adaptation.)

In Section 1.1, it has been suggested that first extending OCaml with resource-friendliness in a backwards-compatible manner can in particular provide a richer playground for tackling the issue of data races in the future. A modest goal can therefore be to organise such a playground such that further drastic evolutions of the language will not be necessary.

If necessary, the hierarchy of polarities can be later be revised so that one can ensure that abstract ownership types guarantee uniqueness. This is currently only the case for concrete types; indeed, any polymorphic function currently is instantiable with a GC type. This was a deliberate choice and showed in a first time a simple and more modest design inspired by polarisation. The alternative is to consider an affine polarity A and set $G <: A$ and $O <: A$ instead of $G <: O$. This refinement would notably allow to consider unrestricted linear in-place update by considering any variable of polarity O an lvalue. For instance, O-polymorphic list concatenation could be implemented in constant time as originally suggested in Lafont (1988).

In the future we would like to have a closer look at the design and uses of Vault (Fähndrich and DeLine, 2002) and Mezzo (Pottier and Protzenko, 2013; Balabonski, Pottier, and Protzenko, 2016). There are synergies with Mezzo, which is also designed around ownership. It explores the design space for a rich type system with fine-grained control of permissions and aliasing using OCaml as a back-end, whereas this proposal focuses on a backwards-compatible resource-management model for OCaml itself. This proposal has less expressive types, even assuming a distinction between linear and read-only borrows à la Rust, but offers lower-level and more expressive abstractions.

14 Summary

"It is time to beat turnstiles into technology. The elegance and efficiency of linear types should allow functional languages to finally go 'main-stream'."

— Baker (1995)

[14]https://github.com/ocamllabs/ocaml-multicore/wiki/Memory-model

This is a proposal for a novel resource-management model compatible with the OCaml philosophy and implementation model, based on a new understanding of RAII, move semantics, and resource polymorphism. It achieves a synthesis of ownership-and-borrowing and linear functional programming, notably by formulating GC allocation as a runtime optimisation for types with trivial destructor.

Details of a language design and of a runtime model are given, requiring little change compared to the current runtime:

- introducing untraced pointers allocated with RAII,
- registering destructors with the exception mechanism, and
- packing destructors with abstract types.

As for types, it proposes a natural polymorphism between ownership, borrow, and GC types. It also raises interesting questions for a safe and practical type system, for which the Cyclone, Rust, and Alms languages provide ample prior work, that will need to be combined.

This proposed extension of OCaml integrates an ownership-and-borrowing model with a tracing GC, with the aim to improve its:

Safety

- New types for resources with destructors can be declared, the destructor is guaranteed to be run predictably and reliably, including in the presence of exceptions.
- Facilities can be provided with default or built-in destructors, such as structures of resources or foreign values.
- Affine closures, notably, are important for local control and effect handlers.
- The Rust model for eliminating data races is likely to apply, and the proposal forms a rich basis with which to further explore issues of control of aliasing in the future.

Efficiency

- Data structures can occasionally be allocated with RAII rather than the GC, offering an allocation method which is timely and reduces GC load, suitable for very large data structures and real-time collection, and for which assignments bypass the write barrier of the GC.
- The implementation for the current OCaml fragment, in particular the GC and the exception handling, is essentially unchanged and therefore remains as efficient as before.
- There are further opportunities for linearity-and lifetime-directed optimisations of garbage collection to investigate (Baker, 1994b; Asati, Sanyal, Karkare, and Mycroft, 2014).

Interoperability

- Foreign values can be received as owned resources, and conversely ownership of values can be given to foreign runtimes.
- The design supports structural mapping between types.

Expressiveness

- Resource polymorphism allows the definition of complex mixtures of resource and non-resource types.
- Resource polymorphism enables resource-generic structures and algorithms.
- It is backwards-compatible with existing OCaml code: polarity tables ensure that the default management for a non-resource data structure is the most generic one, either GC or unboxed.
- It enables efficient control operators.
- The expressiveness of the proposal beyond all its intended uses above remains to be explored.

References

Samson Abramsky. 1993. Computational Interpretations of Linear Logic. *Theor. Comput. Sci.* 111, 1&2 (1993), 3–57. https://doi.org/10.1016/0304-3975(93)90181-R 13

Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the servo web browser engine using Rust. In *ICSE '16*. https://doi.org/10.1145/2889160.2889229 5, 6

Rahul Asati, Amitabha Sanyal, Amey Karkare, and Alan Mycroft. 2014. Liveness-Based Garbage Collection. In *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Albert Cohen (Ed.), Vol. 8409. Springer, 85–106. https://doi.org/10.1007/978-3-642-54807-9_5 55

David F. Bacon, Perry Cheng, and V. T. Rajan. 2004. A unified theory of garbage collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 50–68. https://doi.org/10.1145/1028976.1028982 12

Henry G. Baker. 1994a. Linear logic and permutation stacks - the Forth shall be first. *SIGARCH Computer Architecture News* 22, 1 (1994), 34–43. https://doi.org/10.1145/181993.181999 5, 13, 50, 52

Henry G. Baker. 1994b. Minimum Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures. *SIGPLAN Notices* 29, 9 (1994), 38–43. https://doi.org/10.1145/185009.185016 5, 13, 52, 55

Henry G. Baker. 1995. "Use-Once" Variables and Linear Objects - Storage Management, Reflection and Multi-Threading. *SIGPLAN Notices* 30, 1 (1995), 45–52. https://doi.org/10.1145/199818.199860 5, 13, 33, 52, 53, 54

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The design and formalization of Mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 4 (2016), 14. 54

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6, 6 (1996), 579–612. 5

Nick Benton, Martin Hofmann, and Vivek Nigam. 2016. Effect-dependent transformations for concurrent programs. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, James Cheney and Germán Vidal (Eds.). ACM, 188–201. https://doi.org/10.1145/2967973.2968602 48

Josh Berdine, Peter W O'Hearn, Uday S Reddy, and Hayo Thielecke. 2000. Linearly used continuations. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*. Citeseer, 47–54. 13

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *PACMPL* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093 15, 52

Gavin Bierman. 1995. What is a categorical model of Intuitionistic Linear Logic?. In *Proc. TLCA (Lecture Notes in Computer Science)*, Vol. 902. Springer-Verlag, 78–93. 7

Maximilian C. Bolingbroke and Simon L. Peyton Jones. 2009. Types are calling conventions. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, Stephanie Weirich (Ed.). ACM, 1–12. https://doi.org/10.1145/1596638.1596640 7, 53

Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. 1996. Reference Counting as a Computational Interpretation of Linear Logic. *J. Funct. Program.* 6, 2 (1996), 195–244. https://doi.org/10.1017/S0956796800001660 14

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July*

21-25, 2003, Proceedings (Lecture Notes in Computer Science), Luca Cardelli (Ed.), Vol. 2743. Springer, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9 14, 53

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. https://doi.org/10.1145/286936.286947 5, 14

John Clements. 2006. *Portable and high-level access to the stack with Continuation Marks*. Ph.D. Dissertation. Northeastern University. 45

John Clements and Matthias Felleisen. 2004. A tail-recursive machine with stack inspection. *ACM Trans. Program. Lang. Syst.* 26, 6 (2004), 1029–1052. https://doi.org/10.1145/1034774.1034778 45

Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. 2016. A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. In *Proc. POPL*. https://doi.org/10.1145/2837614.2837652 5

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. 1997. A New Deconstructive Logic: Linear Logic. *Journal of Symbolic Logic* 62 (3) (1997), 755–807. 7

Olivier Danvy and Lasse R. Nielsen. 2005. CPS transformation of beta-redexes. *Inf. Process. Lett.* 94, 5 (2005), 217–224. 42

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *Proc. PLDI 2018*. 54

Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 525–539. https://doi.org/10.1145/3062341.3062357 7, 15, 49

Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 13–24. https://doi.org/10.1145/512529.512532 54

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.), Vol. 3924. Springer, 7–21. https://doi.org/10.1007/11693024_2 9

Edward Gan, Jesse A. Tov, and Greg Morrisett. 2014. Type Classes for Lightweight Substructural Types. In *Proceedings Third International Workshop on Linearity, LINEARITY 2014, Vienna, Austria, 13th July, 2014. (EPTCS)*, Sandra Alves and Iliano Cervesato (Eds.), Vol. 176. 34–48. https://doi.org/10.4204/EPTCS.176.4 15, 16

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102. 7, 13

Jean-Yves Girard. 1991. A new constructive logic: Classical logic. *Math. Struct. Comp. Sci.* 1, 3 (1991), 255–296. 6, 7, 14

Jean-Yves Girard. 1993. On the Unity of Logic. *Ann. Pure Appl. Logic* 59, 3 (1993), 201–217. 7, 14

Timothy G. Griffin. 1990. A Formulae-as-Types Notion of Control. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 47–58. 45

Dan Grossman. 2006. Quantified types in an imperative language. *ACM Trans. Program. Lang. Syst.* 28, 3 (2006), 429–475. https://doi.org/10.1145/1133651.1133653 15, 16

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 282–293. https://doi.org/10.1145/512529.512563 5

Howard E. Hinnant, Peter Dimov, and Dave Abrahams. 2002. A Proposal to Add Move Semantics Support to the C++ Language. (2002). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm 4, 14

Martin Hofmann. 2000. A Type System for Bounded Space and Functional In-Place Update. *Nord. J. Comput.* 7, 4 (2000), 258–289. 15

Gérard P. Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (1997), 549–554. http://journals.cambridge.org/action/displayAbstract?aid=44121 30

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, Carla Schlatter Ellis (Ed.). USENIX, 275–288. http://www.usenix.org/publications/library/proceedings/usenix02/jim.html 5, 14

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154 48

Naoki Kobayashi. 1999. Quasi-Linear Types. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 29–42. https://doi.org/10.1145/292540.292546 15

Yves Lafont. 1988. The linear abstract machine. *Theoretical computer science* 59, 1-2 (1988), 157–180. 13, 54

Peter J. Landin. 1965. *A Generalization of Jumps and Labels*. Technical Report. Subsequently published as Landin (1998). 45

Peter J. Landin. 1998. A Generalization of Jumps and Labels. *Higher-Order and Symbolic Computation* 11, 2 (1998), 125–143. 60

Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language*. Technical Report. INRIA. 42

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Proc. TLCA '99*. 228–242. 7

Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantic Structures in Computation, Vol. 2. Springer. 7

Paul-André Melliès. 2009. *Categorical semantics of linear logic*. Panoramas et Synthèses, Vol. 27. Société Mathématique de France, Chapter 1, 15–215. 15

Paul-André Melliès and Nicolas Tabareau. 2010. Resource modalities in tensor logic. *Ann. Pure Appl. Logic* 161, 5 (2010), 632–653. 53

Naftaly H. Minsky. 1996. Towards Alias-Free Pointers. In *ECOOP'96 - Object-Oriented Programming, 10th European Conference, Linz, Austria, July 8-12, 1996, Proceedings (Lecture Notes in Computer Science)*, Pierre Cointe (Ed.), Vol. 1098. Springer, 189–209. https://doi.org/10.1007/BFb0053062 14

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml - Functional Programming for the Masses*. O'Reilly. 4

Greg Morrisett. 1995. Compiling with Types. (1995). 16, 51

Guillaume Munch-Maccagnoni. 2014. Formulae-as-Types for an Involutive Negation. In *Proceedings of the joint meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS)*. 45

Peter W. O'Hearn, John Power, Makoto Takeyama, and Robert D. Tennent. 1999. Syntactic Control of Interference Revisited. *Theor. Comput. Sci.* 228, 1-2 (1999), 211–252. https://doi.org/10.1016/S0304-3975(98)00359-4 13, 53

François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 173–184. 54

Raphaël Proust. 2016. ASAP: As Static as Possible memory management. (2016). http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-908.pdf 15

Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. 2012. A mechanized semantics for C++ object construction and destruction, with applications to resource management. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 521–532. 4

John C. Reynolds. 1978. Syntactic Control of Interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 39–46. https://doi.org/10.1145/512760.512766 13, 53

Rui Shi and Hongwei Xi. 2013. A linear type system for multicore programming in ATS. *Sci. Comput. Program.* 78, 8 (2013), 1176–1192. https://doi.org/10.1016/j.scico.2012.09.005 15

Arnaud Spiwack. 2014. A dissection of L. (2014). Manuscript. 42

Bjarne Stroustrup. 1994. *The design and evolution of C++*. Pearson Education India. 4

Bjarne Stroustrup. 2001. *Exception Safety: Concepts and Techniques*. Springer Berlin Heidelberg, Berlin, Heidelberg, 60–76. https://doi.org/10.1007/3-540-45407-1_4 4

Bjarne Stroustrup and Herb Sutter. 2015. C++ Core Guidelines. (2015). https://github.com/isocpp/CppCoreGuidelines 6

Bjarne Stroustrup, Herb Sutter, and Gabriel Dos Reis. 2015. A brief introduction to C++'s model for type- and resource-safety. (2015). https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Introductiontotypeandresourcesafety.pdf 4, 5, 16

Mads Tofte and Lars Birkedal. 1998. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20, 4 (1998), 724–767. 5

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value lambda-Calculus using a Stack of Regions. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 188–201. https://doi.org/10.1145/174675.177855 9

Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 447–458. https://doi.org/10.1145/1926385.1926436 15, 16, 31, 34, 42, 52

Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*. North. 15

David Walker. 2005. Substructural type systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, 3–44. 14

Dengping Zhu and Hongwei Xi. 2005. Safe Programming with Pointers Through Stateful Views. In *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10-11, 2005, Proceedings (Lecture Notes in Computer Science)*, Manuel V. Hermenegildo and Daniel Cabeza (Eds.), Vol. 3350. Springer, 83–97. https://doi.org/10.1007/978-3-540-30557-6_8 15