Efficient Deconstruction with Typed Pointer Reversal (abstract)

Guillaume Munch-Maccagnoni, Rémi Douence

Efficient Deconstruction with Typed Pointer Reversal (abstract)

Guillaume Munch-Maccagnoni*
Inria, LS2N CNRS

Rémi Douence†
IMT Atlantique, LS2N CNRS, Inria

8th July 2019

The resource-management model of C++ and Rust relies on compiler-generated destructors called predictably and reliably. In current implementations, the generated destructor consumes stack space proportionally to the depth of the structure it destructs. We describe a way to derive destructors for algebraic data types that consume a constant amount of stack and heap. We discuss applicability to C++ and Rust, and also some implication for anyone wishing to extend an ML-style language with first-class resources.

1. Introduction

Destructors In idiomatic C++[1] and Rust[2], a resource is represented as a value to which a *destructor* is attached: a function called reliably and predictably when a variable goes out of scope, including when an exception is raised (so-called *"Resource Acquisition Is Initialisation"*, see Koenig and Stroustrup, 1990; Stroustrup, 2012). The destructor is a possibly arbitrary side-effecting, non-raising function, and is determined by the type.

For instance, so-called *smart pointers* propose to manage heap allocations automatically and deterministically. Unique pointers[3] do so by ensuring uniqueness of their owner, whereas reference-counting pointers[4] allow for multiple owners. For the latter, the destructor: 1) decrements a reference count, and if the reference count has reached zero, then it 2) *recursively calls the destructor for the contents*, and 3) eventually deallocates the memory[5].

These languages let the user specify the destructor when defining a new type; in fact smart pointers are entirely defined in libraries[6]. The compiler will also derive destructors for compound types.

Movable values with destructor provide an expressive and structured way to manage resources, in particular in the face of errors. Indeed, in comparison to unwind-protect and related resource-management idioms, which it subsumes, it lets the user program with data structures containing resources, and control lifetimes of resources, relaxing for instance the LIFO constraint on the order of release.

We have been studying the applicability of the C++/Rust resource-management model to functional programming, prompted by the understanding that resource management, ensuring the consistency of the state of the world, is a concern that nowadays goes beyond the scope of systems programming.

The stack overflow issue The novice Rust programmer learning to implement simple recursive data structures such as lists and stacks is first shown a simple definition, like the following not unfamiliar to ML programmers[7]:

```
pub struct List<A> { node: Option<Box<(A, List<A>)>>, }
```

They learn that Rust automatically manages memory allocations with Box, and that it knows how to entirely dispose of an allocated list by deriving a destructor that recursively calls the destructor of Box. But soon they are explained that the default destructor will cause a stack overflow on too large lists, and that they ought to redefine it in more efficient imperative style[8]:

```
impl<A> Drop for List<A> {
    fn drop(&mut self) {
        // take ownership of the self.node: replace the contents of
        // self.node with None, and store the previous value in node
        let mut node = std::mem::replace(&mut self.node, None);
        while let Some(mut ptr) = node {
            node = std::mem::replace(&mut ptr.1.node, None);
            // end of ptr's scope:
            // ptr.0 and ptr are implicitly destroyed here
        }
    }
}
```

*Guillaume.Munch-Maccagnoni@inria.fr, Nantes, France
†Remi.Douence@imt-atlantique.fr, Nantes, France

[1] https://isocpp.org/
[2] https://www.rust-lang.org/
[3] std::unique_ptr in C++11 and std::boxed::Box in Rust.
[4] std::shared_ptr in C++11 and std::rc::Rc/std::sync::Arc in Rust.
[5] See for instance https://doc.rust-lang.org/src/alloc/rc.rs.html#828-873, which deals with a weak reference count as well.
[6] With the notable exception of Rust's Box.

[7] The equivalent in C++11 is: `template <typename A> struct List { std::unique_ptr<std::pair<A,List<A>>> node; }`
[8] Inspired from https://rust-unofficial.github.io/too-many-lists/first-drop.html.

This issue with smart pointers in C++ and Rust is known and recurring[9][10][11][12][13][14]. It is the subject of a request for comments for Rust open since 2013[15]. The most detailed publicly available account and acknowledgement of this issue has to our knowledge been given by H. Sutter at CppCon 2016[16], who lists different alternatives to reimplementing the destructor manually in the case of trees: iterate from the bottom, thus trading stack space for traversal time, or enqueue elements to be destroyed, either now or later, in some auxiliary list, thus trading stack space for heap space.

The consensus from these various sources appears to be that the ideal destructor cannot be compiler-generated, and that in general one has to pay either in additional stack space, heap space, or time. Moreover, the reordering or delaying of destructor calls that these solutions require can be harmful when other resources than memory are involved, since this changes the meaning of the program.

On the other end of the memory-management spectrum, built-in mark-and-sweep and reference-counting garbage collectors have since long relied on *pointer reversal* (Schorr and Waite, 1967), a standard graph traversal algorithm which works in linear time and uses no additional space than one available mutable bit within each node—a property that becomes necessary should collection be triggered by an out-of-memory situation, for instance. As a result of our investigations, we have come to derive and generalise pointer reversal from a functional viewpoint, in a form suitable for algebraic data types expressible in C++/Rust. In particular, we call custom effectful destructors for type parameters promptly and in the correct order, and we handle custom smart pointers.

Ordered algebraic data types Problems that appear hopeless at first sometimes require an indirect approach. We have been investigating destructors from the point of view of denotational semantics (Combette and Munch-Maccagnoni, 2018). We obtained the perspective that there is a meaningful interpretation of Rust-style resources in terms of *non-commutative* (or *ordered*) logic, instead of affine or linear logic. We have found this interpretation of linearity closely related to Baker's theses relating system resources to linear logic (Baker, 1994, 1995).

Ordered logic relaxes linear logic by removing the axiom $A \otimes B \cong B \otimes A$, where $\otimes$ is commonly interpreted as the type of pairs, or the comma in typing judgements. The "order" of interest, here, is the one induced by the nesting of scopes—for instance, ordered logic has been used in the past to model the stackability and linearity of exception handlers, and other observations about the stack (Polakow and Yi, 2001; Polakow, 2001).

In hindsight, this ordered phenomenon is clear: in C++ and Rust, the type of pairs ($A \otimes B$) has to specify an arbitrary order in which the components are destructed (say, A then B). Then, the type $B \otimes A$ cannot be contextually isomorphic to the type $A \otimes B$ in general (in the sense of Levy, 2017), since they specify observably distinct destruction orders. This intuition extends to $\otimes$ understood as the comma in contexts: the nesting of scopes determines the order of destruction.

Armed with this perspective, we pondered the notion of *ordered algebraic data types* which naturally arises in the aforementioned denotational setting; that is types:

$$A, B ::= \mathbf{1} \mid X \mid A \otimes B \mid A \oplus B \mid \mu X.A$$

given over parameters $X, Y \ldots$ standing for types supplied with arbitrary side-effecting, non-raising destructors $\delta_X : X \to \mathbf{1}$. Destructors $\delta_A : A \to \mathbf{1}$ for ordered algebraic data types A are defined in function of the destructors δ_X for the type parameters X.

- $\delta_{\mathbf{1}} \stackrel{\text{def}}{=} \lambda().()$
- $\delta_{A \otimes B} \stackrel{\text{def}}{=} \lambda(x, y).\delta_A\, x; \delta_B\, y$
- $\delta_{A \oplus B} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{match}\ x\ \mathsf{with}\ [y.\delta_A\, y \mid z.\delta_B\, z]$
- $\delta_{\mu X.P} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{let\ rec}\ \delta_X\ y\ \mathsf{be}\ \delta_P\, y\ \mathsf{in}\ \delta_X\, x$

For instance, the user wishing to define a type of lists with ordered algebraic data types is faced with a choice between non-isomorphic types $\mathrm{List}(X) \stackrel{\text{def}}{=} \mu Y.1 \oplus X \otimes Y$ and $\mathrm{Tsil}(X) \stackrel{\text{def}}{=} \mu Y.1 \oplus Y \otimes X$ where:

- $\delta_{\mathrm{List}(X)} = \lambda x.\, \mathsf{let\ rec}\ \delta_Y\ y\ \mathsf{be}\ (\mathsf{match}\ y\ \mathsf{with}\ [().() \mid (h, t).\delta_X\, h; \delta_Y\, t])\ \mathsf{in}\ \delta_Y\, x$ (it destructs the elements from head to tail),
- $\delta_{\mathrm{Tsil}(X)} = \lambda x.\, \mathsf{let\ rec}\ \delta_Y\ y\ \mathsf{be}\ (\mathsf{match}\ y\ \mathsf{with}\ [().() \mid (t, h).\delta_Y\, t; \delta_X\, h])\ \mathsf{in}\ \delta_Y\, x$ (it destructs the elements bottom-up).

This presentation says nothing about memory allocation, yet it already displays our problem! Naively, if we have a standard execution model and data representation in mind such as OCaml's, then we are led to think that $\delta_{\mathrm{List}(X)}$ is tail-recursive, whereas $\delta_{\mathrm{Tsil}(X)}$ is not. Yet, $\delta_{\mathrm{Tsil}(X)}$ does admit a tail-recursive implementation: first reverse the $\mathrm{Tsil}(X)$ into a $\mathrm{List}(X)$, and call $\delta_{\mathrm{List}(X)}$. Since we are dealing with linear values, reversing the list can be done in-place, performing no memory allocation.

We paused—nothing in our model appeared to justify such a dissymmetry. We were convinced that there had to be a general solution.

2. Typed pointer reversal

Principle With the previous execution model and data representation in mind, a linear-time, a tail-recursive, non-allocating implementation of δ_A for any ordered algebraic data type A (notwithstanding the cost of parameters δ_X, which can be arbitrary) can be obtained as follows:

[9] (Rust, August 2013) https://github.com/rust-lang/rust/issues/8295
[10] (Rust, February 2015) https://stackoverflow.com/questions/28660362
[11] (C++, January 2015) https://softwareengineering.stackexchange.com/questions/271216
[12] (Rust, July 2016) https://stackoverflow.com/questions/38147453
[13] (Rust, September 2018) https://github.com/orium/rpds/issues/41
[14] (Rust, February 2019) https://github.com/rust-lang/rust/issues/58068
[15] (August 2013) https://github.com/rust-lang/rust/issues/8399, (September 2015) https://github.com/rust-lang/rfcs/issues/686
[16] *"Leak-Freedom in C++... By Default"*, https://www.youtube.com/watch?v=JfmTagWcqoE&t=16m18

1. Consider the definition of δ_A as a naive implementation.
2. Convert it into CPS and defunctionalize the continuations (Danvy and Nielsen, 2001).
3. Observe that before every defunctionalized continuation, one has just made available a memory cell, which can therefore be reused: the memory cell is just large enough, since it already contains all the values the rest of the destructor operates on, plus one element which is now being disposed of, making room for the current continuation.

Example We illustrate this transformation in Appendix A, with an example in OCaml displaying tree-shaped data types and mutually-recursive types, and simulating manual memory deallocation.

Generic formulation We rely on algebraic data types being written as a collection of mutually-recursive sums of products, without nested fixed points, and on the matching data representation where each cell holds the value for one sum of product (with in particular an integer tag containing the information on the variant). With this standard equivalent formulation and uniform representation of algebraic data types, one can easily come up with a generic description of the algorithm and the continuation data structure. Describing the continuation data structure allows one to calculate the size required for the tag of the continuation.

In this abstract, we just report the following observation: the number of different tags can exceed the number of tags for each mutually-recursive type taken separately, but is bounded by their sum. The question of the size of the tag required for typed pointer reversal must therefore be taken into account when determining the representation of mutually-recursive types.

3. Applicability

We report some observations about applicability of this result in Rust, C++ and ML.

3.1. Rust

It is possible to relax the data representation to a more liberal form closer to types that can be expressed in C++/Rust. There, types are unboxed by default and pointers can be used liberally inside the type. We adapted our example in Rust in Appendix B. Since Rust does not support tail recursion, the algorithm is further transformed into a single loop.

To handle types with nested Boxes in general, one can always rewrite them as a set of mutually recursive types where each type corresponds to a pointer's contents, bringing them closer to the previous form. However, doing so shows constraints for the value representation which has to provision enough space for the continuation tag, increasing with the number of distinct types of pointer's contents.

Memory layout Rust's memory layout is less regular than OCaml's, as it attempts to make it as compact as possible. For instance, the representation of the following type:

```
pub enum Tree_2boxes<A,B> {
    Leaf,
    Node(Box<(Tree_2boxes<A,B>,A)>,
         Box<(Tree_2boxes<A,B>,B)>),
}
```

does not hold enough free bits to implement pointer reversal: it misses one available bit. Indeed, Rust recognises that it can test for a null value instead of using a discriminating tag. On the other hand, alignment requirements means the tag field, when present, is often large enough to accommodate additional values. We did not find a formal description of Rust's optimised memory layout, but it is likely that the representation of some types must be reconsidered if one wants to support efficient destructors.

Moreover, since word size and alignment are taken into account, the optimised layout is dependent on the compilation target. Our example from Appendix B is written for a 64-bit target, and we do not know whether it is possible to implement portable efficient destructors by hand. Rust also disables pattern-matching on owned values with custom destructor, when our algorithm is meant as a drop-in replacement of the default one. For all these reasons, some form of compiler support seems highly desirable.

Smart pointers Any native solution will need to take into account the presence of library-defined smart pointers. We have noticed that the algorithm can be adapted for a user-defined smart pointer Ptr as soon as its destructor can be decomposed as follows:

```
fn drop(a: Ptr<T>) {
    if let Some(ptr) = Ptr::try_take_cell(a) {
        unsafe {
            ptr::drop_in_place(ptr.as_ptr());
            Ptr::free_cell(ptr);
        }
    }
}
```

where the following two functions are supplied by the user:

```
fn try_take_cell(a: Ptr<T>) -> Option<ptr::NonNull<T>>
unsafe fn free_cell(a: ptr::NonNull<T>)
```

rather than the destructor being supplied. The resulting algorithm is non-allocating as long as these two functions are non-allocating. This covers at least unique and reference-counting pointers, however this characterisation is open-ended and could cover other cases.

3.2. C++

Unlike Rust, C++ does not have built-in support for algebraic data types, which must be defined in libraries with tagged unions, so it does not make immediate sense with our current description to ask for the compiler to generate efficient destructors. However, unlike Rust, this makes it possible for the user to define portable efficient destructors. Implementing pointer reversal by hand is error-prone, and we hope that our description will already help professionals write correct-by-construction efficient destructors.

3.3. ML

The present result was obtained while exploring the feasibility of a resource[17]-management model for OCaml inspired by the same denotational semantics for types with destructors (Munch-Maccagnoni, 2018). The result is applicable to that model, and is also important for anyone would like to explore similar approaches to static resource types on their own. For instance, we believe that with "linear types", the development of substantial examples where resources are manipulated in data structures, or where one has to handle the failures in the acquisition of resources using error types at scale, would quickly show the need for a distinguished disposal function, for instance implemented with a type class... indeed giving rise to the same issue we aim to address.

In general, the language designer faces a rather large design-space to explore, and the current result restricts this space. Reasoning about first-class resources requires that:

A) a resource is disposed of predictably and reliably, and

B) it is no longer accessed after being disposed of.

One must be careful to state property A) as a safety property (e.g. "the resource is released as soon as the scope ends"), rather than as a liveness property ("there are no leaks"). Resource ownership via values with destructors ensures A). The linear discipline ensures property B), but since it is too constrained, usability requires means to relax it (Odersky, 1992, and many others). Then, enforcing B) can become challenging.

Here we do not mean to explore or prescribe a particular technique to achieve B), but to make a general observation on the design space. In a language in which all memory is handled with a tracing garbage collector, it was not clear whether property B) had to hold for all ordered algebraic data types, or only base resource types.[18] Concretely, the language designer can be tempted to explore a design where data structures containing resources are allowed to be accessed after the destructor ran, as long as the actual resources at the leaves are not accessed (via some permissive and perhaps simpler form of borrowing). But the present algorithm based on reusing cells shows that both properties A) and B) have to be be ensured for any resource, including compound ones, unless one is ready to give up efficient destructors (which, for the functional programming style, means having to find another way to avoid likely stack overflows).

As it turns out, these are the same conditions that allows one to experiment with static "out-of-heap" memory allocation for those resource data structures, everything else remaining equal.[19]

4. Related works

Pointer reversal (Schorr and Waite, 1967) is thought to be tricky to get right and to prove correct. Once again, our favourite program transformation that brings to the surface hidden semantics of the programming language—the defunctionalization of continuations (Danvy and Nielsen, 2001; Reynolds, 1972)—gives rise to a systematic and correct-by-construction solution. This methodology of using continuation semantics for program optimisation in a series of refinement steps was pioneered by Wand (1980) and is by now well established. The key property that cell sizes match between data and continuations has previously been noted by Sobel and Friedman (1998) in a dual context. In that paper, they proposed to implement data-structure-generating functions efficiently by *"recycling continuations"* into data, whereas we recycle data into continuations. They further conjectured: *"since anamorphisms and catamorphisms are duals we wonder whether there is a dual to our technique, which might provide some sort of optimization for catamorphisms."*

The solution, especially our initial example of List and Tsil, reminds of the zipper (Huet, 1997). As Huet notes, the zipper itself can be considered a *"disciplined use of pointer reversal techniques"* (Huet, 2003). The similarity between zippers and defunctionalized continuations has been noted by Danvy (2008). We initially sought a solution in the theory of zippers and type derivatives (McBride, 2001). In the end, the off-the-shelf technique to reflect the semantics in the syntax was more readily applicable and brought an immediate solution.

With the current result, we directly derive pointer reversal from defunctionalization for its initial purpose, garbage collection. We do not expect it to be very surprising in hindsight, but we believe it to be useful and important to be aware of in the context of resource management.

Acknowledgements

We thank the anonymous reviewers from the ML 2019 workshop. We thank Leo White for thought-provoking discussions about the resource-management model, and Gabriel Scherer for pointing us to the relevant garbage collection literature.

[17] Resources can also refer to space and time bounds (as investigated in Resource Aware ML for instance). We see it as a different application of linear logic, and we use the terms "resource management" and "first-class resources" to refer specifically to resources as values that are hard to copy or dispose of.

[18] An observation we owe to L. White.

[19] Outside of language-design motivations, static allocation solves two implementation-specific obstacles to an application to OCaml shown by the current work: the obstacle that rewriting tags breaks the multicore GC invariants, and the obstacle that the tag is too small (8 bits) if one has to follow the layout expected by the GC.

References

Henry G. Baker. 1994. 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 2

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 2

Guillaume Combette and Guillaume Munch-Maccagnoni. 2018. A resource modality for RAII. In *LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages* (2018-04-16). https://hal.inria.fr/hal-01806634 2

Olivier Danvy. 2008. Defunctionalized Interpreters for Programming Languages. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP 2008)*. 131–142. 4

Olivier Danvy and Lasse R. Nielsen. 2001. *Defunctionalization at Work*. Research Report BRICS RS-01-23. Department of Computer Science, Aarhus University, Aarhus, Denmark. Extended version of an article presented at the Third International Conference on Principles and Practice of Declarative Programming (PPDP 2001), Firenze, Italy, September 5-7, 2001. 3, 4

Gérard Huet. 1997. The zipper. *Journal of functional programming* 7, 05 (1997), 549–554. 4

Gérard Huet. 2003. *Linear Contexts, Sharing Functors: Techniques for Symbolic Computation*. Springer Netherlands, Dordrecht, 49–69. https://doi.org/10.1007/978-94-017-0253-9_4 4

Andrew Koenig and Bjarne Stroustrup. 1990. Exception Handling for C++. In *Proceedings of the C++ Conference. San Francisco, CA, USA, April 1990*. 149–176. 1

Paul Blain Levy. 2017. Contextual isomorphisms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 400–414. 2

Conor McBride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts. (2001). 4

Guillaume Munch-Maccagnoni. 2018. Resource Polymorphism. (2018). https://arxiv.org/abs/1803.02796 4

Martin Odersky. 1992. Observers for linear types. In *ESOP '92*. Springer. https://doi.org/10.1007/3-540-55253-7_23 4

Jeff Polakow. 2001. *Ordered Linear Logic and Applications*. Ph.D. Dissertation. 2

Jeff Polakow and Kwangkeun Yi. 2001. Proving Syntactic Properties of Exceptions in an Ordered Logical Framework. In *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings (Lecture Notes in Computer Science)*, Herbert Kuchen and Kazunori Ueda (Eds.), Vol. 2024. Springer, 61–77. https://doi.org/10.1007/3-540-44716-4_4 2

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of 25th ACM National Conference*. Boston, Massachusetts, 717–740. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword Reynolds (1998). 4

John C. Reynolds. 1998. Definitional Interpreters Revisited. *Higher-Order and Symbolic Computation* 11, 4 (1998), 355–361. 5

H. Schorr and W. M. Waite. 1967. An Efficient Machine-independent Procedure for Garbage Collection in Various List Structures. *Commun. ACM* 10, 8 (Aug. 1967), 501–506. https://doi.org/10.1145/363534.363554 2, 4

Jonathan Sobel and Daniel P. Friedman. 1998. Recycling Continuations. *SIGPLAN Not.* 34, 1 (Sept. 1998), 251–260. https://doi.org/10.1145/291251.289452 4

Bjarne Stroustrup. 2012. Foundations of C++. In *European Symposium on Programming (ESOP 2012)*. Springer, 1–25. 1

Mitchell Wand. 1980. Continuation-Based Program Transformation Strategies. *J. ACM* 27, 1 (1980), 164–180. 4

A. Example in OCaml

Full runnable source code for OCaml 4.06 available here: https://gitlab.com/gadmm/efficient-drops-demo/blob/master/abtree.ml.

```
module type Droppable = sig
  type t
  val drop : t -> unit
end

(* In our example we simulate manual deallocation *)
let free x = ()

(* In our example we simulate testing for moving, as would be needed
   for instance when dropping what remains of a value after
   pattern-matching. *)
let moved x = false

(* The following example illustrates:
   - Tree-shaped data types
   - Mutually-recursive types
 *)

type ('a, 'b) atree =
  | ALeaf
  | ATree of ('a, 'b) btree * 'a * ('a, 'b) btree
and ('a, 'b) btree =
  | BLeaf
  | BTreeA of ('a, 'b) atree * 'b
  | BTreeB of ('a, 'b) btree * 'b

module ABTree (A : Droppable) (B : Droppable)
  : Droppable with type t = (A.t, B.t) atree
= struct
  type at = (A.t, B.t) atree
  type bt = (A.t, B.t) btree
  type t = at

  (* The definition of drop taken as a naive implementation, where in
     addition we show that we can handle two additional features of a
     realistic implementation:
      - we test for moving, as if it could have been moved,
      - we manually deallocate the cell, as if it was statically allocated.
     *)
  let rec drop_naive x =
    if not (moved x) then
      match x with
      | ALeaf -> ()
      | ATree (bt1, a, bt2) as cell -> (
          (* free bt1 *)
          drop_naive_b bt1 ;
          A.drop a ;
          (* free bt2 *)
          drop_naive_b bt2 ;
          free cell
        )
  and drop_naive_b x =
    if not (moved x) then
      match x with
      | BLeaf -> ()
      | BTreeA (at, b) as cell -> (
          (* free at *)
          drop_naive at ;
          B.drop b ;
          free cell
        )
      | BTreeB (bt, b) as cell -> (
          (* free bt *)
          drop_naive_b bt ;
          B.drop b ;
          free cell
        )

  (* drop_naive converted into CPS *)
  let rec drop_naive_cps x k =
    if moved x then
      k ()
    else
      match x with
      | ALeaf -> k ()
      | ATree (bt1, a, bt2) as cell -> (
          drop_naive_b_cps bt1 @@ fun () ->
          A.drop a ;
          drop_naive_b_cps bt2 @@ fun () ->
          free cell ;
          k ()
        )
  and drop_naive_b_cps x k =
    if moved x then
      k ()
    else
      match x with
      | BLeaf -> k ()
      | BTreeA (at, b) as cell -> (
          drop_naive_cps at @@ fun () ->
          B.drop b ;
          free cell ;
          k ()
        )
      | BTreeB (bt, b) as cell -> (
          drop_naive_b_cps bt @@ fun () ->
          B.drop b ;
          free cell ;
          k ()
        )

  (* defunctionalisation *)
  type drop_cont =
    | Top
    | KA1 of at * drop_cont * A.t * bt
    | KA2 of at * drop_cont
    | KB1 of bt * drop_cont * B.t
    | KB2 of bt * drop_cont * B.t

  let rec drop_naive_defunc x k =
    if moved x then
      return k
    else
      match x with
      | ALeaf -> return k
      | ATree (bt1, a, bt2) as cell -> drop_naive_b_defunc bt1 (KA1 (cell, k, a, bt2))
  and return = function
    | Top -> ()
    | KA1 (cell, k, a, bt2) -> A.drop a ; drop_naive_b_defunc bt2 (KA2 (cell, k))
    | KA2 (cell, k) -> free cell ; return k
    | KB1 (cell, k, b) -> B.drop b ; free cell ; return k
    | KB2 (cell, k, b) -> B.drop b ; free cell ; return k
  and drop_naive_b_defunc x k =
    if moved x then
      return k
    else
      match x with
      | BLeaf -> return k
      | BTreeA (at, b) as cell -> drop_naive_defunc at (KB1 (cell, k, b))
      | BTreeB (bt, b) as cell -> drop_naive_b_defunc bt (KB2 (cell, k, b))

  (* now we are going to re-use the cell:
     ATree becomes KA1 becomes KA2 becomes freed
     BTreeA becomes KB1 becomes freed
     BTreeB becomes KB2 becomes freed
  *)

  type drop_cont_reused =
    | Top
    | KA1 of { k : drop_cont_reused ; a : A.t ; bt2 : bt }
    | KA2 of { k : drop_cont_reused ; void : unit ; void2 : unit }
    | KB1 of { k : drop_cont_reused ; b : B.t }
    | KB2 of { k : drop_cont_reused ; b : B.t }

  type _ tag =
    | TagKA1 : at tag
    | TagKA2 : drop_cont_reused tag
    | TagKB1 : bt tag
    | TagKB2 : bt tag

  (* purple magic! re-use a cell *)
```

```
  let as_k : type a. a tag -> a -> drop_cont_reused -> drop_cont_reused =
    fun i x k ->
    let x = Obj.repr x |> Sys.opaque_identity in
    Obj.set_field x 0 (Obj.repr k) ;
    Obj.set_tag x (Obj.magic i) ;
    Obj.obj x

  let as_ka1 ~(cell:at) = as_k TagKA1 cell
  let as_ka2 ~(cell:drop_cont_reused) = as_k TagKA2 cell
  let as_kb1 ~(cell:bt) = as_k TagKB1 cell
  let as_kb2 ~(cell:bt) = as_k TagKB2 cell

  let rec drop_reuse_defunc x k =
    if moved x then
      return k
    else
      match x with
      | ALeaf -> return k
      | ATree (bt1, a, bt2) as cell -> drop_reuse_b_defunc bt1 (as_ka1 ~cell k)
  and return = function
    | Top -> ()
    | KA1 { k ; a ; bt2 } as cell ->
       A.drop a ;
       drop_reuse_b_defunc bt2 (as_ka2 ~cell k)
    | KA2 { k ; _ } as cell -> free cell ; return k
    | KB1 { k ; b ; _ } as cell -> B.drop b ; free cell ; return k
    | KB2 { k ; b ; _ } as cell -> B.drop b ; free cell ; return k
  and drop_reuse_b_defunc x k =
    if moved x then
      return k
    else
      match x with
      | BLeaf -> return k
      | BTreeA (at, b) as cell -> drop_reuse_defunc at (as_kb1 ~cell k)
      | BTreeB (bt, b) as cell -> drop_reuse_b_defunc bt (as_kb2 ~cell k)

  let drop x = drop_reuse_defunc x Top
end

(* Test *)

module Int : Droppable with type t = int
  = struct
  type t = int
  let drop int = Printf.printf "Dropping: %u\n" int
end

let example_value =
  let b3 = BTreeA (ALeaf, 5) in
  let a2 = ATree (BLeaf, 6, BTreeB(BLeaf, 7)) in
  let a1 = ATree (b3, 4, BLeaf) in
  let b1 = BTreeA (a1, 1) in
  let b2 = BTreeB (BTreeA(a2, 3), 2) in
  ATree (b1, 0, b2)

module M = ABTree (Int) (Int)

let _ = M.drop example_value
```

B. Example in Rust

Full runnable source code for Rust 1.31 available here: https://gitlab.com/gadmm/efficient-drops-demo/blob/master/abtree.rs. The program has to make assumptions about the memory representation; we tested it with compilation target `x86_64-unknown-linux-gnu`.

```
// Demo of efficient drops
// rustc -O abtree.rs && ./abtree

use std::mem;
use std::ptr;


// set to 100000 to overflow the stack with
// the stock destructor
static N: i64 = 100000;


#[derive(Debug)]
pub enum ATree<A,B>
where A: std::fmt::Debug, B: std::fmt::Debug {
    ALeaf,
    ALeaf2, // prevent "Option" optim
    ATree(Box<BTree<A,B>>, A, Box<BTree<A,B>>),
}

#[derive(Debug)]
pub enum BTree<A,B>
where A: std::fmt::Debug, B: std::fmt::Debug {
    BLeaf,
    BTreeA(Box<ATree<A,B>>, A),
    BTreeB(Box<BTree<A,B>>, B),
}

#[derive(Debug)]
enum State {
    DropA,
    DropB,
    Return
}

#[allow(dead_code)]
#[derive(Debug)]
enum DropCont<A,B>
where A: std::fmt::Debug, B: std::fmt::Debug {
    Top,
    KA1(*mut DropCont<A,B>, A, Box<BTree<A,B>>),
    KA2(*mut DropCont<A,B>),
    KB(*mut DropCont<A,B>, B),
}

#[allow(dead_code)]
#[derive(Debug)]
enum Tag {
    Top,
    KA1,
    KA2,
    KB,
}

unsafe fn as_k<A,B,T>(tag: Tag, cell: *mut T, k: *mut DropCont<A,B>)
                      -> *mut DropCont<A,B>
where A: std::fmt::Debug, B: std::fmt::Debug {
    let cell: *mut [*mut (); 2] = mem::transmute(cell);
    (*cell)[0] = mem::transmute(tag as u64);
    (*cell)[1] = mem::transmute(k);
    mem::transmute(cell)
}

unsafe fn take_cell<T>(t: &mut Box<T>) -> *mut () {
    mem::transmute(Box::into_raw(ptr::read(t)))
}

unsafe fn reverse<A,B,T>(x: &mut *mut (),
                         tag: Tag,
                         new_x: &mut Box<T>,
                         k: &mut *mut DropCont<A,B>)
where A: std::fmt::Debug, B: std::fmt::Debug {
    let cell = mem::replace(x, take_cell(new_x));
    *k = as_k(tag, cell, *k);
}

unsafe fn pop_k<A,B>(k: &mut *mut DropCont<A,B>,
                     next_k: &mut *mut DropCont<A,B>)
                     -> *mut DropCont<A,B>
where A: std::fmt::Debug, B: std::fmt::Debug {
    ptr::replace(k, ptr::read(next_k))
}

unsafe fn free_cell_as<A,B,T>(val: T, cell: *mut DropCont<A,B>)
where A: std::fmt::Debug, B: std::fmt::Debug {
    let a: *mut T = mem::transmute(cell);
    ptr::write(a, val);
```

```
        Box::from_raw(a);
    }

fn drop<A,B>(x: ATree<A,B>)
where A: std::fmt::Debug, B: std::fmt::Debug {
    // hypotheses on the memory layout
    assert_eq!(8, mem::size_of::<A>());
    assert_eq!(8, mem::size_of::<B>());
    let mut top = DropCont::Top;
    let mut k = &mut top as *mut DropCont<A,B>;
    let mut state = State::DropA;
    unsafe {
        //always allocated with Box
        let mut x: *mut () = mem::transmute(Box::into_raw(Box::new(x)));
        unsafe fn read_as<T>(x: *mut ()) -> *mut T { mem::transmute(x) };
        loop {
            match state {
                State::DropA => match &mut *read_as::<ATree<A,B>>(x) {
                    ATree::ALeaf | ATree::ALeaf2 =>
                        state = State::Return,
                    ATree::ATree(bt1, _, _) => {
                        state = State::DropB;
                        reverse(&mut x, Tag::KA1, bt1, &mut k);
                    },
                },
                State::DropB => match &mut *read_as::<BTree<A,B>>(x) {
                    BTree::BLeaf =>
                        state = State::Return,
                    BTree::BTreeA(at, _) => {
                        state = State::DropA;
                        reverse(&mut x, Tag::KB, at, &mut k);
                    },
                    BTree::BTreeB(bt, _) => {
                        reverse(&mut x, Tag::KB, bt, &mut k);
                    },
                },
                State::Return => match &mut *k {
                    DropCont::Top => return,
                    DropCont::KA1(k2, a, bt2) => {
                        ptr::drop_in_place(a);
                        state = State::DropB;
                        x = take_cell(bt2);
                        k = as_k(Tag::KA2, k, ptr::read(k2));
                    },
                    DropCont::KA2(k2) => {
                        free_cell_as(ATree::ALeaf::<A,B>, pop_k(&mut k, k2));
                    },
                    DropCont::KB(k2, b) => {
                        ptr::drop_in_place(b);
                        free_cell_as(BTree::BLeaf::<A,B>, pop_k(&mut k, k2));
                    },
                }
            }
        }
    }

// Test

#[derive(Debug)]
struct Int { val: i64, }

impl Drop for Int {
    fn drop(&mut self) {
        //println!("Dropping {}", self.val)
    }
}

fn int(i: i64) -> Int {
    Int{val: i}
}

fn example_value(a: ATree<Int,Int>) -> ATree<Int,Int> {
    let b3 = BTree::BTreeA(Box::new(a), int(5));
    let a2 = ATree::ATree(Box::new(BTree::BLeaf), int(6),
                          Box::new(BTree::BTreeB(Box::new(BTree::BLeaf), int(7))));
    let a1 = ATree::ATree(Box::new(b3), int(4), Box::new(BTree::BLeaf));
    let b1 = BTree::BTreeA(Box::new(a1), int(1));
    let b2 = BTree::BTreeB(Box::new(BTree::BTreeA(Box::new(a2), int(3))), int(2));
    ATree::ATree(Box::new(b1), int(0), Box::new(b2))
}

fn main() {
    fn bench(drop_fun: fn(ATree<Int,Int>) -> (), print: bool) {
        let mut x: ATree<Int,Int> = ATree::ALeaf;
        for _i in 0..N {
            x = example_value(x);
        }
        let now = std::time::SystemTime::now();
        drop_fun(x);
        if print {
            println!("{:?}", now.elapsed().expect("Time went backwards"));
        }
    };
    bench(drop, false);
    println!("Custom drop with re-use");
    bench(drop, true);
    println!("Rust's compiler-generated drop");
    bench(mem::drop, true);
}


// Epilogue:

// This can be adapted to other smart pointers, provided they provide
// try_take_cell and free_cell functions that decompose their drop as
// follows. The resulting algorithm is non-allocating as long as the
// two functions are non-allocating.

#[allow(dead_code)]
struct Ptr<T> { a: std::marker::PhantomData<T> }

#[allow(dead_code)]
impl<T> Ptr<T> {
    fn try_take_cell(_a: Ptr<T>) -> Option<ptr::NonNull<T>> { None }

    unsafe fn free_cell(_a: ptr::NonNull<T>) { panic!("not mine!") }

    fn drop(a: Ptr<T>) {
        if let Some(ptr) = Ptr::try_take_cell(a) {
            unsafe {
                ptr::drop_in_place(ptr.as_ptr());
                Ptr::free_cell(ptr);
            }
        }
    }
}
```