Efficient Deconstruction with Typed Pointer Reversal
(abstract)
Guillaume Munch-Maccagnoni, Rémi Douence

To cite this version:

HAL Id: hal-02177326
https://inria.hal.science/hal-02177326v2
Submitted on 9 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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\(^7\):\(^8\):

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

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 } } }
```

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 } } }
```

\(^1\)https://isocpp.org/
\(^2\)https://www.rust-lang.org/
\(^3\)std::unique_ptr in C++11 and std::box::Box in Rust.
\(^4\)std::shared_ptr in C++11 and std::rc::Re::Rs::Sync 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\textsuperscript{9,10,11,12,13,14,15,16}. It is the subject of a request for comments for Rust open since 2013\textsuperscript{15}. The most detailed publicly available account and acknowledgement of this issue has to our knowledge been given by H. Sutter at CppCon 2016\textsuperscript{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 effective 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 $A \otimes B \equiv B \otimes A$, 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 $(A \otimes B)$ to have 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:

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

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

- $\delta_1 \equiv \lambda().$
- $\delta_{A \otimes B} \equiv \lambda(x, y).\delta_A x \times \delta_B y$
- $\delta_{A \oplus B} \equiv \lambda x. \text{match } x \text{ with } [y.\delta_A y \mid z.\delta_B z]$
- $\delta_{\mu X. P} \equiv \lambda x. \text{let rec } \delta_X y \equiv \delta_P y \text{ in } \delta_X x$

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 $\text{List}(X) \equiv \mu Y.1 \oplus X \otimes Y$ and $\text{Tsl}(X) \equiv \mu Y.1 \oplus X \otimes X$ where:

- $\delta_{\text{Tsl}(X)} \equiv \lambda x. \text{let rec } \delta_x y \equiv \text{match } y \text{ with } [(h, t)](l) \text{ in } \delta_y h \text{ in } \delta_X t$ (it destructs the elements from head to tail),
- $\delta_{\text{Tsl}(X)} \equiv \lambda x. \text{let rec } \delta_x y \equiv \text{match } y \text{ with } [(h, t)](l) \text{ in } \delta_x h \text{ in } \delta_X t$ (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 $\delta_{\text{Tsl}(X)}$ is tail-recursive, whereas $\delta_{\text{Tsl}(X)}$ is not. Yet, $\delta_{\text{Tsl}(X)}$ does admit a tail-recursive implementation: first reverse the $\text{Tsl}(X)$ into a $\text{List}(X)$, and call $\delta_{\text{Tsl}(X)}$. 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 $\delta_A$ for any ordered algebraic data type $A$ (notwithstanding the cost of parameters $\delta_X$, which can be arbitrary) can be obtained as follows:
1. Consider the definition of \( \delta_A \) 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:

```rust
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 \( \text{Ptr} \) as soon as its destructor can be decomposed as follows:

```rust
fn drop(a: \text{Ptr}<T>) {
    if let Some(ptr) = \text{Ptr}::\text{try\_take\_cell}(a) {
        unsafe {
            \text{ptr}::\text{drop\_in\_place}(ptr.as\_ptr());
            \text{Ptr}::\text{free\_cell}(ptr);
        }
    }
}
```

where the following two functions are supplied by the user:

```rust
fn \text{try\_take\_cell}(a: \text{Ptr}<T>) \Rightarrow \text{Option}<ptr::\text{NonNull}<T>>
unsafe fn \text{free\_cell}(a: ptr::\text{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\textsuperscript{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.\textsuperscript{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 leave 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.\textsuperscript{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 Tsl, 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.

\textsuperscript{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.

\textsuperscript{18}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 multcore GC invariants, and the obstacle that the tag is too small (8 bits) if one has to follow the layout expected by the GC.

\textsuperscript{19}An observation we owe to L. White.
References


A. Example in OCaml

B. Example in Rust

Full runnable source code for Rust 1.31 available here: https://gitlab.com/gadm/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.

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

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

// set to 100000 to overflow the stack with
let example_value =

let b3 = BTreeA(Leaf, 5) in
let a2 = BTree(BLeaf, 6) in
let a1 = BTree(BLeaf, 7) in
let b1 = BTree(A(3, 4)) in
let b2 = BTree(B(A(2, 3), 2)) in

module M = ABTree(int, int)

let () = M.drop example_value

unsafe fn as_k<A,B,T>(tag: Tag, cell: *cell) -> DropCont<A,B,T> =

#derive(Debug)
pub enum⟨A,B⟩

where A: std::fmt::Debug,
    B: std::fmt::Debug {
    ALeaf, // prevent "Option" optim
    BTreeA(A(3, 4)),
    BTreeB(B(A(2, 3), 2)),
    }.

#derive(Debug)
pub enum⟨A,B⟩

where A: std::fmt::Debug (B: std::fmt::Debug {
    BLeaf, BTreeA(BTreeA(A(3, 4))), BTreeB(BTreeB(BLeaf, 7)),
    }, B)

#derive(Debug)
enum⟨A,B⟩

where A: std::fmt::Debug (B: std::fmt::Debug {
    BLeaf, BTreeA(BTreeA(BLeaf, 7)),
    })

#derive(Debug)
enum⟨A,B⟩

where A: std::fmt::Debug (B: std::fmt::Debug {
    BLeaf, BTreeA(BTreeA(BLeaf, 7)),
    })

// the stock destructor
static N: i64 = 100000;

// set to 100000 to overflow the stack with
```

```
let mut k = &

let mut state = State::DropA;

match

let mut state = State::Return;

match

let mut state = State::DropB;

match

let mut state = State::DropA;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;

match

let mut state = State::Return, 

match

let mut state = State::DropB;