Modular efficient deconstruction with typed pointer reversal

Jean Caspar, Guillaume Munch-Maccagnoni

Modular efficient deconstruction with typed pointer reversal

Jean Caspar[1] and Guillaume Munch-Maccagnoni[2]

[1]École normale supérieure – PSL university
[2]INRIA, LS2N, Nantes

Abstract

Destructors, responsible for releasing memory and other resources in languages such as C++ and Rust, can lead to stack overflows when releasing a recursive structure that is too deep. In certain cases, it is possible to generate an efficient destructor (non-allocating and tail recursive) using a typed variant of pointer reversal. We extend this technique by making it more modular, in order to handle abstract types, separate compilation, and unboxed types.

1 Introduction

In some programming languages, such as Rust and C++, memory management is performed by the compiler via the insertion of some code at the end of the scope of values, responsible for releasing the memory allocated on the heap. Such code is called a destructor. Since they run predictably and reliably when variables get out of scope, destructors can be used to reason about resources other than memory, such as files, shared data protected by a mutex, network connections, or transactions. The use of destructors as a resource-handling mechanism is called RAII (*Resource Acquisition Is Initialization*) [KS90].

The stack overflow problem In RAII-based languages, destructors recursively call the destructors of each sub-field; therefore, if the type they operate on is recursive, then the destructor should be recursive as well. In the evolution of this model, as incarnated by the programming languages C++11 and Rust, compiler-generated destructors themselves are potentially recursive (via "smart pointers"), and their naive implementations by contemporary compilers potentially cause stack overflows. It was believed that one cannot implement destructors in general without allocating on the stack or on the heap, or without changing the order and time of destruction, as Herb Sutter explained [Sut16]. The fact that generated destructors themselves can overflow the stack makes it a problem of compiler correctness.

Typed pointer reversal Given that the programmer can supply an arbitrary non-raising function acting as a destructor on a per-type basis, generated destructors must take programmer-provided destructors into account, and are specific to a given type.

Continuing previous work by Douence and the second author [MMD19], the subject of this paper is to derive efficient and correct destructors for recursive types, starting from their specifications (that is, their naive implementations) and applying provably-correct

transformations. As shown previoulsy, it is indeed possible to obtain implementations which are tail-recursive and which do not allocate, by starting from the naive implementation and applying standard transformations: namely a continuation-passing style (CPS) transformation followed by defunctionalization [Rey72, Wan80, DN01], and a transformation to re-use the memory of the (unaliased) value as a way to store the defunctionalized continuations [Laf88, Bak92]. A dual transformation, about recycling continuations into data, had also been observed earlier by Sobel and Friedman [SF98]. In effect, this transformation amounts to a typed generalization of pointer reversal [SW67], a graph traversal algorithm developed for garbage collection, notorious for being hard to get right and to reason about.

In particular, the resulting implementation is equivalent to the naive implementation, but it does not overflow the stack. (Another benefit which we can expect is slightly improved performance due to better cache locality, but we do not propose benchmarks in this paper.)

This paper assumes familiarity with neither CPS nor defunctionalization. We directly give the general shape of the resulting implementation below in Section 2, which we use as a starting point.

Limitations to its applicability & goals The typed pointer reversal method suffers from limitations if we want to use it in the construction of compilers:

- It does not handle abstract types: if a custom destructor supplied for a type is recursive, then it is not clear how to apply the transformation. Therefore, this method cannot be used by the compiler to automatically derive efficient destructors.
- It does not handle separate compilation: if the specification of a destructor is unknown, even if its efficient implementation is provided, then we cannot derive another efficient destructor that calls this destructor, because the method is not compositional.
- It does not handle unboxed types: currently, it assumes that all values are boxed behind a pointer (Lisp-like representation of values), whereas unboxed values (C-like representation of values, as used in Rust and C++) are not handled.

In this paper, we show that is is possible to extend typed pointer reversal by making it modular, in order to handle abstract types and separate compilation (Section 3), and we sketch an application to unboxed types (Section 5).

Since pointer reversal is tricky to implement by hand and to prove correct, we ideally want to prove that the new transformations themselves are correct, and maybe relate them to ones that have already been studied; as far as the present work is concerned this is work in progress.

Througout the paper, we propose examples and descriptions in an ML-like syntax. There are two reasons for this choice:

1. the Lisp-like memory representation is simple, so is easy to start with, and there are additional issues with the C-like memory representation of Rust and C++ that we will not touch upon before the later parts of the paper;
2. it is possible to imagine extensions of ML that incorporate RAII-based resource-management [MM23]. In this case, the challenges go beyond C++ and Rust, as witnessed by examples from Section 4 involving elaborate forms of polymorphism.

Example: B-trees The B-tree data structure constitutes a good example of the limitations of the approach from the previous work. A B-tree is a tree where each node can have a non-bounded number of children, and holds a non-bounded number of key-value pairs. It can be defined in ML as follows:

```
type 'a btree = Leaf of (int * 'a) array | Node of (int * 'a) array * 'a btree array
```

This type uses arrays—values whose size is not known during compilation. This is not

supported by typed pointer reversal as we described so far, since `array` is not an algebraic data type. In addition, an efficient destructor for `array` would be of no help in letting us deal with a recursion occurring via its parameter as in its third occurrence above. In fact, the naive destructor for arrays, which iterates on the elements, is already an efficient destructor (since `array` is not recursive on its own).

Furthermore, we want to be able to derive an efficient destructor for B-trees, even when the implementations of arrays and of its destructor are opaque (for instance opaque to the compiler, when arrays are defined in another compilation unit). This might actually be of a lesser concern in languages that are not so strict about separate compilation, such as C++ and Rust. Lastly, in languages such as C++ and Rust, where we would substitute vectors for arrays, the vectors would be unboxed (as triplets consisting of a size, a capacity, and a pointer to a backing array).

All in all, if we want to apply typed pointer reversal to the implementation of programming languages, we need to determine how to handle abstract types, separate compilation, and unboxed types.

2 On efficient destructors for algebraic data types

2.1 Generic shape for typed pointer reversal

To start from where Douence and the second author [MMD19] left off, we first describe a generic form of the efficient destructors obtained with their method, for algebraic data types of the following generic shape:

$$A_i = B_{i,1}(A_1, \ldots, A_n) + \cdots + B_{i,m_i}(A_1, \ldots, A_n) \qquad 1 \le i \le n$$
$$B_{i,j}(A_1, \ldots, A_n) = C_{i,j,1} \times \cdots \times C_{i,j,l_{i,j}} \qquad 1 \le i \le n, 1 \le j \le m_i$$

where $C_{i,j,k}$ is either of the form $A_{i'}$, or some other type which does not contain any $A_{i'}$. In the latter case, the destructors are assumed given (as `drop_C_{i,j,k}`), and possibly arbitrary.

The specification of the destructor is to traverse the structure depth-first, from left to right, calling the destructors of the $C_{i,j,k}$, and freeing the memory of each node it exits—as follows:

```
(* for all i: *)
let rec drop_A_i t = match t with
(* for all j: *)
| B_{i,j}(c_1, ..., c_l) -> drop_C_{i,j,1} c_1; ...; drop_C_{i,j,l} c_l; free t
```

Above, recursion takes place when $C_{i,j,k}$ is of the form $A_{i'}$.

Applying the CPS transformation yields tail-recursive implementation of destructors, with an accumulator (the continuation), so as to avoid consuming stack space. Then, through defunctionalization, the continuation becomes a first-order data structure. We call these implementations of destructors `drop_iter_A_i`. Then, `drop_A_i` is equivalent to `drop_iter_A_i` called with initial continuation.

The type of defunctionalized continuations has the form $\text{cont} = 1 + \sum_{i,j,k} K^k_{i,j,1} \times \cdots \times K^k_{i,j,l_{i,j}}$ where the sum is indexed by all tuples (i,j,k) where $C_{i,j,k}$ is of the form $A_{i'}$ for some i', and where $K^k_{i,j,k'} = \text{cont}$ whenever $k' \le k$ and $C_{i,j,k}$ is of the form $A_{i'}$, and $K^k_{i,j,k'} = C_{i,j,k}$ otherwise.

After the transformation, we obtain the functions from Listing 1 below, written in ML-like pseudo-code plus explicit freeing of memory. We write `K_{i,j,k}` the constructor of the variant corresponding to (i,j,k) in the sum cont, and `Empty` the constructor for 1. In addition, we added the annotation `@` on constructors to denote that a memory cell can be re-used, as explained next.

```
(* for all i: *)
let rec drop_iter_A_i t cont = match t with
(* for all j: *)
| B_{i,j}(c_1, ..., c_l) ->
    (* We statically find the least k such that C_{i,j,k} is of the form A_{i'}. *)
    (* If k exists: *)
    drop_C_{i,j,1} c_1; ... drop_C_{i,j,k-1} c_{k-1};
    drop_iter_A_{i'} c_k (K_{i,j,k}@t (cont, c_{k+1}, ..., c_l))
    (* If k does not exist: *)
    drop_C_{i,j,1} c_1; ... drop_C_{i,j,l} c_l; free t; invoke cont
| ...

and invoke = function
(* for all i, j and k: *)
| K_{i,j,k}(cont, c_{k+1}, ..., c_l) as m ->
    (* We statically find the least k' > k such that C_{i,j,k'} is of the form
       A_{i'}. *)
    (* If k' exists: *)
    drop_C_{i,j,k+1} c_{k+1}; ... drop_C_{i,j,k'-1} c_{k'-1};
    drop_iter_A_{i'} c_{k'} (K_{i,j,k'}@m (cont, c_{k'+1}, ..., c_l))
    (* If k' does not exist: *)
    drop_C_{i,j,k+1} c_{k+1}; ... drop_C_{i,j,l} c_l; free k; invoke cont
| ...
| Empty -> ()

let drop_A_i a = drop_iter_A_i a Empty
```

Listing 1. A generic efficient destructor for algebraic data types.

The last part of the transformation involves re-using memory to store the continuations, so as to avoid allocating on the heap. We use the notation `K@v(a, b, ...)` above to mean that `K(a, b, ...)` can be allocated by reusing the memory cell of the value `v`. This involves mutating both the fields and the tag of `v`.

There are two key observations to make here. First, notice that `t` is no longer used in the rest of the destructor; it also cannot be used later in the program, since the destructor eventually frees its memory. Second, the new continuation always consists of a strict suffix of the previous value, plus a field to store the previous continuation. In particular, the re-used value is large enough, and at most one field needs to be mutated.

As written down, it is manifest that this implementation is tail-recursive and does not allocate.

2.2 Possible improvements

The previous section assumes that there is no limit on the number of variants. But there can be more variants for the continuation than there is in the original type. In practice, this can introduce additional constraints for the memory representation if values.

In order to bound the number of variants, the first observation is that we can reduce the amount of variants for the type cont to at most two for each pair (i, j). Indeed, from the second field being destroyed onwards, a second cell is available, which can store an additional integer. The k distinct variants can thus be replaced by two variants plus an auxiliary integer ranging from 2 to k.

We can further extend this solution to recursive types containing arrays (i.e. some of the $C_{i,j,k}$ being arrays possibly containing some A_is). As above, after the first element has been destroyed, we store an integer in the first field, which indicates how many fields have been destroyed so far.

2.3 Example: B-trees

Having just explained how to extend the method to arrays, let us go back to the example of B-trees. In Listing 2 we provide a destructor for a simplified version of the B-tree. The keyword `cast` is used below to coerce the type of values in an unsafe way. We need two variants in `cont` in order to deal with the array. The first one, `K`, allows us to drop its first element, and the store the continuation at its place. The second one, `K'`, takes advantage of the fact that there is at least two free memory cells and store in the second one the number of cells already dropped. The variants `Leaf` and `Node` contain exactly zero or one element, thus we can further optimize and not adding variants for them inside `cont`.

```
type btree = Leaf | Node of btree array
type cont = Empty | K of cont | K' of cont * int

let rec drop_btree t k = match t with
| Leaf -> invoke k
| Node x -> free t; drop_array x k
and drop_array a k =
    if Array.length a = 0 then invoke k
    else let y = a.(0) in drop_btree y (K@a k)
and invoke = function
| Empty -> ()
| K k' as k ->
    let (x : btree array) = cast k in
    if Array.length x = 1 then invoke k'
    else let y = x.(1) in drop_btree y (K'@x (k', 2))
| K' (k', i) as k ->
    let (x : btree array) = cast k in
    if Array.length x = i then invoke k'
    else let y = x.(i) in drop_btree y (K'@x (k', i + 1))
```

Listing 2. Efficient B-tree destructor.

3 An interface for a modular efficient drop

While we have shown that it is possible to handle B-trees by extending the algorithm to arrays, the goal is to obtain a modular interface to handle types that could be defined as an abtract type by a library (e.g. vector in C++/Rust), perhaps in an opaque compilation unit.

3.1 An interface in ML-like languages

In order to handle abstract types, we propose a modular interface that any type can implement in order to provide an efficient destructor. It allows composing such destructors automatically for algebraic data types, and can be implemented by hand for more complex data types. This (unsafe) interface is given in Figure 1b in the syntax of ML modules. It can be seen as both as a description of the interface needed by a compiler to generate an efficient destructor, and a refinement of the Drop *trait* in Rust's terminology (essentially the module type in Figure 1a).

The `Drop` interface allows destructing a type `t`, and must recursively call the destructor of each of the values it contains. As we have seen so far, an implementation of `Drop` can be derived automatically for all mutually-recursive types comprised of algebraic types and arrays.

The `DropIter` interface allows implementing and composing efficient destructors; it is unsafe because the parameter `'a` represents a type of continuations, but it is challenging to describe this type in advance.

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

(a) Drop interface.

```
module type Base = sig val base : int end

module type DropIter = sig
    type t
    val nb_state : int
    module Make(B : Base) : sig
        type cont
        val drop_iter : t -> cont -> unit
        val invoke : cont -> unit
    end
end
```

(b) DropIter interface.

Figure 1. Interfaces for destructors as ML modules.

The key idea behind this interface is that each implementation will declare `nb_state` variants of a `cont` recursive type (which will never be formally declared), with these variants having a tag belonging to the interval [`base`; `base` + `nb_state`[. The implementation is parameterized by the `base` value, to be chosen such that the various implementations of `DropIter` involved in the destruction of a data structure do not interfere with each other.

The values `nb_state` and `base` are therefore constant and known at compile-time. The functions `drop_iter` and `invoke` play the same roles as in the previous sections, and can be automatically derived for algebraic data types. Their implementation for any `base` is known at compile-time.

Lastly, the implementation of a `DropIter` is generally parametrised by the `DropIter`s of its type parameters: the modular efficient drop interface for a parametrised type (`'a t`) is a functor from `DropIter` to `DropIter`.

The type `cont` represents a type of continuations, which depends on the values of `nb_state`, `base` and the other instances of `DropIter` involved. We have not yet correctly typed the continuation in this interface in any language in a way that respects the required representation of values, we relied on unsafe features instead.

In `invoke`, implementations should look at the tag of the parameter, and if it is some universal and fixed integer constant, it should return (this means the iteration reached the end). If it belongs to the interval [`base`; `base` + `nb_state`[, it should handle it appropriately (either by doing something directly or by dispatching it to a `DropIter` implementation instantiated by the current implementation). Lastly, if it belongs to one of the `DropIter` implementations by which this implementation is parameterized, it must dispatch it accordingly. In correct implementations, other cases do not happen.

Given a `DropIter` instance, one can build a `Drop` instance in the following way:

```
module DropIterToDrop(D: DropIter) : (Drop with type t = D.t) = struct
    type t = D.t
    module Impl = D.Make(struct let base = 0 end)
    let drop x = Impl.drop_iter x Empty
end
```

Here, `Empty` is the constant on which all `invoke` functions must return.

3.2 Example with a generic type

Now let us focus on a concrete example. Suppose we have a type parameterized by a type variable, for which we want to implement `DropIter`, such as the following:

```
type 'a tree = Node of 'a tree * 'a * 'a tree | Leaf
```

The type parameter is required to also have a modular efficient destructor, therefore the implementation of `DropIter` for `'a tree` is parametrised by a `DropIter` for `'a`. In the

pseudo-code below, we use ellipses to denote that the tags of variants are chosen among $[\mathsf{base}; \mathsf{base}+2[$.

```
module DropIterTree (D: DropIter) : (DropIter with type t = D.t tree) = struct
    type t = D.t tree
    let nb_state = 2
    module Make(B : Base) = struct
        let d_base = B.base + nb_state
        module DImpl = D.Make(struct let base = d_base end)
        type cont = Empty
        | ...
        | KDropVal of cont * D.t * D.t tree (* with tag offset B.base *)
        | KDropRight of cont * D.t tree
        | ...

        let rec drop_iter t k = match t with
        | Leaf -> invoke k
        | Node (l, x, r) -> drop_iter l (KDropVal@t (k, x, r))
        and invoke k = match k with
        | Empty -> ()
        | KDropVal (k',x,r) -> D.drop_iter x (KDropRight@k (k',r))
        | KDropRight (k',r) -> free k; drop_iter r k'
        | K _ when d_base <= tag K && tag K < d_base+D.nb_state -> DImpl.invoke k
        | _ -> assert false
    end
end
```

Listing 3. Efficient and modular drop implementation for `'a tree`.

The `KDropVal` variant represents a node whose left child has been dropped. The `KDropRight` variant represents a node whose left child and the value it holds have been dropped, but not the right child.

This functor can be automatically derived with the typed pointer reversal method (e.g. by the compiler) in the case of mutually algebraic data types and arrays, similarly to the version of section 2.

4 Limitations due to polymorphism

Nested or non-regular data types are recursive polymorphic data types that can recursively instantiate themselves with different parameters.

Common examples of such data types, as presented by Matthes [Mat08], include:

- Perfect binary trees : $\mathrm{Tree}\ \alpha = \alpha + \mathrm{Tree}(\alpha \times \alpha)$
- Bushes : $\mathrm{Bush}\ \alpha = 1 + \alpha \times \mathrm{Bush}(\mathrm{Bush}\ \alpha)$
- Lambda calculus with de Bruijn indexes and explicit substitutions:
  $\mathrm{Lam}\ \alpha = \alpha + \mathrm{Lam}\ \alpha \times \mathrm{Lam}\ \alpha + \mathrm{Lam}(1+\alpha) + \mathrm{Lam}(\mathrm{Lam}\ \alpha)$

If we try to apply our algorithm to derive an efficient destructor for such data types, we will be stuck because our algorithm relies on the fact that we encounter a finite number of types in the implementation of a single recursive destructor. But if we want to generate a destructor for perfect binary trees $\mathrm{Tree}\ \alpha$, we will need to generate the destructor for $\mathrm{Tree}(\alpha \times \alpha)$, $\mathrm{Tree}((\alpha \times \alpha) \times (\alpha \times \alpha))$, and so on. This would lead to an infinite number of states, at least with this approach.

But, for such types T we can still define a less efficient destructor which is better than the naive one, by using the *drop* function for β when we try to destruct $T\ \beta$ with $\alpha \neq \beta$. The

stack consumption will then increase only when, while destructing a value of type $T\ \alpha$, we destruct a value of type $T\ \beta$ with $\beta \neq \alpha$; outside of that, it is constant. Therefore for a value of type $T\ \alpha$, the stack usage is proportional to the maximum number of edges $T\ \beta \to T\ \gamma$ where possibly $\beta \neq \gamma$ that we encounter in a path from the origin to a leaf.

This limitation does not apply to languages like C++ or Rust where all polymorphic arguments are monomorphized at compile-time, because we cannot define recursive functions on such a type. Monomorphization (substituting all generic types by a concrete type) is indeed less expressive than polymorphism à la ML.

5 Deconstructing unboxed types

5.1 Refining the interface for unboxed types

Currently, we have only considered values that are boxed, that is, are accessed through an indirection. But some languages such as C++ and Rust allow someone to manipulate unboxed values, that are laid out consecutively in memory. A problem arises when dealing with unboxed values, as we cannot overwrite the unboxed value with the continuation before we have dropped it entirely. It worked with boxed values because we have been exchanging values of the same size.

We propose in the next section a refinement of `DropIter` which drops a value of type `T` placed at a specific offset of a type `U`, rather than accessed by indirection. We deal with transitive subfields in the same way, in terms of offsets of a parent boxed type. Furthermore, a problem arises when dealing with mutually recursive types naively: if we have two mutually recursive types `T` and `U` of different unboxed sizes, then it is not clear which should be the size of the continuation.

We can address this problem by using tagged pointers for the continuation, where the tag is used to identify the continuation types before we dereference them. In typical 64-bit architecture, many bits of pointers are non-significant and can be re-used.[1] We call this kind of tagged pointer to a continuation a `ContPointer`.

5.2 Unboxing the efficient destructor

We introduce a refinement of the previous interface, with explicit pointers, in order to drop a type `T` that is inlined in another type `U`, at offset o (in bytes):

$$\texttt{drop\_field}_{\texttt{T},o,\texttt{U}} : \texttt{U}* \to \texttt{ContPointer} \to 1$$

$$\texttt{invoke\_field}_{\texttt{T},o,\texttt{U}} : \texttt{ContPointer} \to 1$$

$$\texttt{retrieve} : \texttt{T}* \to \texttt{ContPointer}$$

where `U*` is the type of pointers of `U`.

We can derive an implementation of this interface called $\texttt{DropField}_{\texttt{T},o,\texttt{U}}$ for a type `T` starting with a `DropIter` interface for `T`, under a few conditions: we require that all calls to `drop_iter` for an inlined `T` take a continuation of the form `KRest@u(k, ...)` where `u` is the pointer to `U` that has been recycled (that is, we do not optimize the last call of a sequence of destructors as we did in subsection 2.2). We call this transformation *unboxing*.

The function `drop_field` is essentially `drop_iter`, but where right at the beginning we retrieve a `T*` by doing pointer arithmetic with the `U*` pointer in argument and the offset o. We use this `T*` pointer as in `drop_iter`, except that we store the tag in the pointer `U*` instead of the pointer `T*`; we still recycle the memory of our `T*` pointer (since we do not know the state of the rest of the structure, we can only access a portion of it).

[1] For instance, on the x86_64 architecture, one can re-use at least the lower 3 bits due to alignment, and at least 7 of the upper bits if 5-level paging is used [Int23, Vol. 3A, §4.5].

For `invoke_field`, we cast the `ContPointer` into a `U*` and also do pointer arithmetic to retrieve a `T*`. On it, we execute the same code that `invoke_iter` (the function called `invoke` formerly) executes.

But inside these two functions we change one thing: we do not free the memory used by the pointer `T*`. Furthermore, the last recursive call which calls the continuation passed as a parameter (of the form `k = KRest(k', ...)`) is changed to a call to $\texttt{invoke\_iter}_{\texttt{U}}\ \texttt{KEnd}_{\texttt{T},o,\texttt{U}}(\texttt{k}, ...)$ where we recycle the `U*` pointer and store the continuation k somewhere in the current structure, and where $\texttt{KEnd}_{\texttt{T},o,\texttt{U}}$ is a new state that belongs to `U`.

And that is where the `retrieve` function comes in: it must be able to retrieve this continuation after the the destruction of the nested `T*`. The code to handle a $\texttt{KEnd}_{\texttt{T},o,\texttt{U}}$ is also inserted into the $\texttt{invoke\_iter}_{\texttt{U}}$ function:

```
invoke_iter_U (KEnd_T_o_U as k) =
    let u = (cast k : U*) in
    let t = (cast (u + o) : T*) in
    let k' = retrieve t in
    (* this code can be inlined and the variant KRest deleted *)
    invoke_iter_U (KRest@u(k', ...))
```

We need also to change the code calling `drop_iter` and `invoke_iter` for `T` when it is a field of `U`.

Every call to $\texttt{drop\_iter}_{\texttt{T}}\ \texttt{t}\ \texttt{KRest@u}(\texttt{k}, ...)$ where `t` is inlined in some structure `U` must be replaced with $\texttt{drop\_field}_{\texttt{T},o,\texttt{U}}\ \texttt{u}\ \texttt{k}$ where o is the difference in bytes between `t` and `u`, and every call to $\texttt{invoke\_iter}_{\texttt{T}}\ k$ must be replaced with $\texttt{invoke\_field}_{\texttt{T},o,\texttt{U}}\ k$. When several types are inlined inside one another, the type `U` and the offset must be taken from the outermost type, that is, from the pointer that currently owns the allocation for the whole structure.

5.3 Example: handling vectors

The vector in C++ and Rust is an unboxed type which consists of a pointer to the start of a backing array, a capacity pointer to its end, and a length pointer that points past the last element of the vector.

If the capacity is greater than the length, then we can store the continuation at the end of the backing array. We can decrement the length pointer and drop the next element through `drop_iter` if there remain elements to destroy, or free the backing array and call the continuation otherwise. If the capacity is equal to the length, then one can first store the continuation by overwriting the capacity (since this information is redundant), while tagging the length pointer to indicate this first step taken. After the first step, the capacity pointer is restored from the length pointer, there is now room at the end, and one proceeds as above.

All this obviously relies on implementation details of vectors. The implementation of the efficient destructor therefore needs to be given by standard library implementors, via the modular efficient drop interface.

Further code and examples, including a detailed implementation for C++ vectors, are available in the public repository https://gitlab.com/JeanCASPAR/artefacts-jfla-2023.

6 Further work

More work is needed to formally define and prove the correctness of our transformations, especially the unboxing transformation. We would like to find more conceptual explanations of our transformations, which might simplify proofs of correctness. We would also like to clarify the constraints on the memory representation of variants, and distinguish pertinent memory representation details from irrelevant ones.

References

[Bak92] Henry G Baker. Lively linear lisp: "look ma, no garbage!". *ACM Sigplan notices*, 27(8):89–98, 1992.

[DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at Work. Research Report BRICS RS-01-23, Department of Computer Science, Aarhus University, Aarhus, Denmark, July 2001.

[Int23] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide*, September 2023. Volume 3.

[KS90] Andrew Koenig and Bjarne Stroustrup. Exception handling for C++. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 149–176, San Francisco, CA, USA, 1990.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical computer science*, 59(1-2):157–180, 1988.

[Mat08] Ralph Matthes. Recursion on nested datatypes in dependent type theory. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, pages 431–446, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[MM23] Guillaume Munch-Maccagnoni. Resource polymorphism: A proposal for integrating first-class resources into ML (abstract). In *ML Family workshop (ML'23)*, 2023.

[MMD19] Guillaume Munch-Maccagnoni and Rémi Douence. Efficient deconstruction with typed pointer reversal (abstract). In *ML Family workshop (ML'19)*, 2019.

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.

[SF98] Jonathan Sobel and Daniel P. Friedman. Recycling continuations. *SIGPLAN Not.*, 34(1):251–260, September 1998.

[Sut16] Herb Sutter. Leak-Freedom in C++... By Default. `https://www.youtube.com/watch?v=JfmTagWcqoE&t=?t=978`, 2016. CppCon 2016.

[SW67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[Wan80] Mitchell Wand. Continuation-Based Program Transformation Strategies. *J. ACM*, 27(1):164–180, 1980.