Utrecht University

Computing Science Master Thesis

Reference Counting with Reuse in Roc

Author:
J. Teeuwissen, BSc
(6456944)

Supervisors:
F. de Vries, MSc[1]
Dr. W. Swierstra[2]
Prof. Dr. G. Keller[2]
Dr. T. McDonell[2]

[1]Roc Programming Language Foundation
[2]Utrecht University

August 10, 2023

Abstract

Most functional programming languages use a tracing garbage collector to automatically reclaim unused memory. But tracing garbage collection reclaims garbage memory at an unspecified time, which results in stop-the-world pauses, an increase in peak memory usage, and the inability to perform in-place mutations. Instead, a reference counting garbage collector can be used. Having the exact reference count of a structure allows for in-place mutation and the immediate reclamation of unused memory, at the cost of runtime overhead. The reference counting and reuse algorithm *Counting Immutable Beans* from Ullrich and de Moura [18] attempts to reduce this overhead by borrowing references. However, their reuse implementation can lead to an arbitrary increase in peak memory usage. The reference counting algorithm *Perceus* from Reinking et al [1] opts for precise reference counting instead, resulting in garbage free programs. Programs that can be further improved by cancelling out matching opposite reference counting operations with *drop specialisation* and with the *drop guided reuse* algorithm from Lorenzen and Leijen [2]. Despite both reuse algorithms showing excellent performance for simple programs, they are unable to fully utilise reuse opportunities across join points. We use Roc, a purely functional programming language, to compare the previous Counting Immutable Beans implementation to an extended version of Perceus with drop guided reuse. We show that our new implementation decreases reference counting overhead, increases memory reuse, and is competitive with functional programming languages that use tracing garbage collection.

Contents

1 Introduction **3**
- 1.1 Research Questions . . . 3
- 1.2 Contributions . . . 3

2 Overview **4**
- 2.1 Roc . . . 4
  - 2.1.1 Projections . . . 4
  - 2.1.2 Non-RC values . . . 4
  - 2.1.3 Normalisation and Defunctionalisation . . . 4
  - 2.1.4 Joinpoints . . . 5
- 2.2 Reference Counting . . . 6
  - 2.2.1 Atomic Reference Counting . . . 6
  - 2.2.2 Counting Immutable Beans . . . 6
  - 2.2.3 Perceus . . . 8
- 2.3 Reuse Analysis . . . 8
  - 2.3.1 Drop Guided Reuse . . . 9
- 2.4 Specialisation . . . 10
  - 2.4.1 Drop Specialisation . . . 10
  - 2.4.2 Drop Guided Drop Specialisation . . . 11
  - 2.4.3 Reuse Specialisation . . . 12

3 Formalisation **12**
- 3.1 Calculus . . . 13
- 3.2 Heap Semantics . . . 13
- 3.3 Reference Counting . . . 15
- 3.4 Reuse Analysis . . . 18
- 3.5 Drop Guided Drop Specialisation . . . 21

4 Results **23**
- 4.1 Operations . . . 24
- 4.2 Performance . . . 24

5 Conclusion **26**
- 5.1 Discussion . . . 26
- 5.2 Future Work . . . 27

References **28**

1 Introduction

Computer programs use the computer heap-based memory to temporarily store data. When a program requests to store data, it (or the operating system) allocates a portion of memory to store this data in. But memory is a finite resource, meaning that unused memory should be deallocated to be used again later. Tracing garbage collection (also known as Mark & Scan or Mark & Sweep) is a common technique for automatically reclaiming such unused memory during program execution. Tracing GC works in cycles of two separate stages. A cycle can be started periodically or when there is not enough free memory available for a new allocation. The first stage starts in *roots* of the program (like the stack, registers or static variables) and recursively traverses the reachable memory structures and marks these structures. The second stage scans all the allocated memory and reclaims those that are not marked. Several variations exist to reduce fragmentation, cycle frequency and cycle duration [4][11][13]. A benefit of tracing garbage collection is that cyclic references are garbage collected like any structure, making it suitable for languages like Haskell and OCaml that allow the creation of cyclic data structures through laziness or mutable data types. However, the garbage memory in-between cycles cause a higher memory footprint than strictly necessary, which can negatively affect performance. But more importantly: memory is reclaimed at an unspecified time, which makes it impossible to immediately reuse allocated memory for new structures when no longer in use, resulting in additional expensive allocations. An alternative memory management strategy is reference counting garbage collection (RC), a technique where each heap allocated structure is accompanied by a count of the pointers that point to the structure. The reference count is updated when a reference to the structure is added or removed, e.g. by passing it to two functions. When the reference count drops to zero, meaning the structure is no longer referenced, the memory can be deallocated or reused for a new structure, as will be discussed in section 3.4. But reference counting does have downsides. The reference count itself has to be stored in memory, increasing the size of each structure. And the execution of the reference counting operations themselves can be quite time consuming, especially when incrementing and decrementing the reference count when not strictly necessary, an important consideration in the algorithms we will discuss in section 2.2. Additionally, it is difficult to deal with cycles in structures. When two structures have pointers to each other, the reference count of neither will ever drop to zero. Thus, both structures will stay in memory forever [3]. In this work we adapt the *Perceus* reference counting algorithm and *drop specialisation* optimization from Reinking et all [1] and the *drop guided reuse* algorithm from Lorenzen and Leijen [2] to be compatible with Roc, a purely functional programming language. And we compare the new algorithms with the previous reference counting and reuse algorithms based on *Counting Immutable Beans* from Ullrich and de Moura [18] in terms of runtime and memory performance.

1.1 Research Questions

The questions we try to answer in this paper are:

- In what way do the Perceus reference counting algorithm and the drop-guided reuse algorithm need to be modified, in order to be compatible with the Roc compiler and drop specialisation?
- What are the effects of the adapted Perceus reference counting, drop-guided reuse, and drop specialisation algorithms on the performance of Roc programs when compared with the Counting Immutable Beans reference counting and reuse algorithm?

1.2 Contributions

We make the following contributions to answer these questions:

- We extend both the resource calculus and programmatic implementation for Perceus and drop guided reuse to work with join points and projections, including a new syntax-directed resource rule-set for reuse analysis.
- We describe a new approach for drop specialisation called *drop guided drop specialisation*, which takes the core concept from Perceus: Garbage Free Reference Counting with Reuse from Reinking et al and improves its effectiveness and versatility.
- We implement the new Perceus reference counting, drop guided reuse, and drop guided drop specialisation in Roc. And compare both the static and dynamic performance characteristics of the new algorithms with the existing Counting Immutable Beans reuse and reference counting algorithm.

2 Overview

We will start our overview with a summary of the Roc programming language. We will look at the basic syntax and relevant compiler features. After which we will take a closer look at reference counting, reuse analysis and other optimisations as described in other papers.

2.1 Roc

Roc is "A fast, friendly, functional language" [17] with strict evaluation, referential transparency, and no automatic currying. Fast compilation and performant applications are an explicit goal [16]. The goal of this section is to get familiar with Roc and to show compiler features that require adaptations to the algorithms discussed in section 2.2 to 2.4. The examples used in this section have statements inserted by the compiler highlighted in gray, to make a clear distinction between what a user can write and what the compiler might generate.

2.1.1 Projections

Roc uses product and sum types to represent data. Product types, called records in Roc, allow multiple values (fields) to be wrapped in another value, and for these fields to be projected out. For example:

```
Tuple a b : {l: a, r: b}

fst : Tuple a b -> a
fst = \t -> t.l
```

Listing 1: Record Example

Sum types are called tag unions in Roc. They can be one of multiple values. For example:

```
Maybe a : [Just a, Nothing]

mapMaybe : Maybe a, (a -> b) -> Maybe b
mapMaybe = \maybe, f -> when maybe is
    Just a -> Just (f a)
    Nothing -> Nothing
```

Listing 2: Tag Example

The `when ... is` pattern match from the previous example is removed during compilation, generating code similar to this for `mapMaybe`:

```
mapMaybe = \maybe, f ->
    maybeTagId = getTagId maybe
    switch maybeTagId:
        case 0:
            a = maybe.Just.0
            Just (f a)
        default:
            Nothing
```

Listing 3: Tag Example

In which the pattern match is replaced by a switch on the id of the tag. The original bound variable `a` is now explicitly retrieved using a projection `Just.0`, the first value given that the tag is `Just`. This transformation simplifies the logic and makes further compilation easier. We will add projections in general to later formalisations to be more precise and better correspond with Roc.

2.1.2 Non-RC values

Roc makes a distinction between reference counted and not reference counted values. The memory size of integers, records, and non-recursive tags (like maybe or result) is known at compile time, which allows these values to be allocated on the stack to reduce allocations and improve performance. The size of lists, strings, and recursive tags (like linked lists or trees) cannot be determined at compile time. Hence these structures are allocated on the heap. We opt to ignore this distinction because the check whether a value is reference counted or not is straightforward to implement and is inconsequential to the formalisation from section 3. But for consistency we will use types that *are* reference counted in the examples.

2.1.3 Normalisation and Defunctionalisation

The IR of Roc is both normalised and defunctionalized. The normalisation of a program means to translate nested expression into multiple, chained, simpler ones. See listings 4 and 5 for an normalisation example for the multiplication of complex numbers.

```
Complex a : { r: Int a, i: Int a}

multiplyComplex :
    Complex a, Complex a
    -> Complex a
multiplyComplex =
    \{r: xr, i: xi}, {r: yr, i: yi}
    -> {r: xr * yr - xi * yi,
        i: xr * yi + xi * yr}
```

Listing 4: Normalization Example

```
multiplyComplex = \x, y ->
    xr = x.r
    xi = x.i
    yr = y.r
    yi = y.i
    a = xr * yr
    b = xi * yi
    c = a - b
    d = xr * yi
    e = xi * yr
    f = d - e
    {r: c, i: f}
```

Listing 5: Normalization Example Cont

In this example we see the complex computations inside the constructor transformed into simple operations. First all fields are projected out and saved into an intermediate variable. Then the actual computations are performed on single variables. After which the results are stored in a record. This simplifies the IR by removing the need to model nested expressions and specifies a clear order in which the operations are performed. This in turn simplifies the remaining steps in compilation, like the generation of assembly.

Defunctionalisation, similar to normalisation, is used to simplify the IR. Defunctionalisation replaces higher order functions with specialised first order functions. For example, if we define a function to apply another function on both elements of the tuple defined before:

```
applyTuple : Tuple a a, (a -> b)
    -> Tuple b b
applyTuple = \{l, r}, f
    -> {l: f l, r: f r}

applyTuple {l: 0, r: 1} \x -> x * 2
```

Defunctionalisation (without normalisation) will in turn generate the following code:

```
applyTuple : Tuple I64 I64
    -> Tuple I64 I64
applyTuple = \{l, r}
    -> {l: double l, r: double r}

double : I64 -> I64
double = \x -> x * 2

applyTuple {l: 0, r: 1}
```

The specific types enables analyses to know what the layout of each argument/variable is within the function, which is crucial for e.g. reuse analysis. The defunctionalisation used by Roc is based on the work from William Brandon, Benjamin Driscoll, Frank Dai, Wilson Berkow, and Mae Milano [5].

2.1.4 Joinpoints

The previous sub section mentioned the normalisation of programs and the benefits thereof. We, however, did not yet explain what normalisation for complex expressions looks like. For example:

```
makeLoudCatDogSound : Bool -> Str
makeLoudCatDogSound = \isDog ->
    sound = if isDog
        then "Woof"
        else "Miauw"
    Str.concat sound "!"
```

The function from the example above checks the boolean parameter. If `isDog` is true, the "Woof" sound branch is taken, otherwise the "Miauw" branch. Afterwards the branches *join* back up to use string concatenation to add an exclamation mark. A naive way to normalise this function would be to add the continuation, in this case the assignment and the concatenation, to both branches. As shown in the example below:

```
makeLoudCatDogSound = \isDog ->
    if isDog
        then
            sound = "Woof"
            Str.concat sound "!"
        else
            sound = "Miauw"
            Str.concat sound "!"
```

This transformation gives us a normalized program. But this approach combined with a larger continuation or multiple nested expressions could result in an code size explosion. A different approach would be to put the `if then else` expression in a function, and to call this function with the `isDog` parameter to obtain its sound. But transforming this kind of logic into function calls has different downsides:

- Function calls require additional stack and register manipulation, making them more expensive than simple branching.
- Moving code into functions can break further optimisations. Reuse analysis for example does not work across functions.

A solution to this problem is *join points*[12]. Join points are a combination of *joins* that define arguments with a continuation and *jumps* that can jump to this continuation.

Using join points the same example would look like the following:

```
makeLoudCatDogSound = \isDog ->
    joinpoint soundJp = \sound ->
        Str.concat sound "!"
    if isDog
        then jump soundJp "Woof"
        else jump soundJp "Miauw"
```

Listing 6: Joinpoint Example Cont

These joinpoints can be nested and jumps can be recursive. Jumps can only occur at the end of a statement, guaranteeing that the jump can be compiled into a simple jump (or goto). Any variables in scope before the definition of the join (its closure) can be used without passing them as a parameter. Another benefit of join points is that they can be used to solve a multitude of problems like:

- Joining back the control flow after branching, like in the example above.
- Performing the same action under different conditions. Like a pattern match having multiple matching conditions for the same branch.
- Tail call elimination: Some recursive functions can be optimised (tail call elimination or tail call optimisation) using join points so that a jump can be performed instead of a recursive call. Saving on stack frames and function calls.

Without requiring special IR constructs to deal with each one separately or duplicating code, which is why join points see usage in languages like Haskell, Lean, and Roc. And because Roc heavily relies on these join points, they should be considered when designing and implementing RC and reuse algorithms. It would be sub-optimal for reuse analysis to fail for a function as simple as `makeLoudCatDogSound`.

2.2 Reference Counting

Tracing (generational) garbage collection has been the de facto garbage collection mechanism for functional programming languages. But the benefits and drawbacks change in the context of strict pure functional languages with an explicit control flow, like Roc or Koka. Drawbacks like requiring special pointers to reclaim cyclic structures becomes a non-issue, since cyclic structures cannot be created. While benefits like being able to manipulate structures in-place becomes even more important if the alternative is copying the entire structure. And an explicit control flow allows structures to be dropped immediately after their latest usage by not being bound to the unwinding of the stack, resulting in a lower memory usage and more optimisation opportunities. These benefits are why we will focus on RC GC for the remainder of this paper. Reference counting in its simplest form is performed using two operations: *dup* and *drop* (also known as *inc* and *dec*). Dup increases the reference count of a structure by one whilst drop lowers it by one. If a unique (a reference count of exactly one) reference is *dropped*. It will drop all nested references and free the structure's memory instead. The goal of a reference counting algorithm is to insert these operations in the IR of a language such that memory is kept allocated while still in use but deallocated when not. While, preferably, inserting as few operations as possible. We will look at two algorithms in section 2.2.2 and 2.2.3. But first: some context.

2.2.1 Atomic Reference Counting

In addition to the inserting the dup and drop operations at the right places, their implementation requires consideration as well. In concurrent environments it is possible for structures to be shared across multiple threads. Thus, every step in the RC operations should be performed atomically, to prevent multiple concurrent writes from resulting in an erroneous count. Performing these steps atomically however, can increase their cost significantly [19]. This problem can be partially remedied by using a lock-free approach or by reducing the use of atomic reference counting in general through *Biased Reference Counting* [7]. But the exact mechanism used to modify reference counts is not the focus of this paper, and as such we will restrict ourselves to the *effect* of reference counting operations.

2.2.2 Counting Immutable Beans

The previous tracing GC algorithm used by Roc is based on the work by Ullrich and de Moura called *Counting Immutable Beans* [18]. The goal of the described algorithm is to reduce the number of inserted reference counting operations by marking function parameters as either owned or borrowed at compile time. A borrowed parameter is assumed to be kept alive by the caller, therefore the caller does not have to dup the reference count of a variable before passing it to the function and the callee does not have to drop it. Reducing the total number of reference count operations. Counting Immutable Beans is performed in three steps:

1. Pattern matches and constructors are paired as possible memory reuse opportunities.

2. The borrow signature of all functions is determined, using the reuse information.

3. The reference counting operations are inserted, using the borrow signature of other functions.

Reuse analysis is performed using a heuristic-based algorithm to recursively iterate over a function body, while keeping track of variables that get pattern matched on (scrutinised) and their memory size in their respective case arms. (Note that the paper uses the arity of the matched constructor to reflect the size in memory, because the values in the constructor are just pointers to another value in the heap and thus of equal size. But that this is not always the case for Roc where values can be contained inside the structure directly as well and as such would require keeping track of the memory layout specifically). And using this information to find later constructions of a matching size. If found, the analysis will insert a `reset` statement after the latest usage of the scrutinee and a `reuse` statement before the construction. A reset returns the pointer of the variable if the reference count is unique, or a special token (typically a null pointer) otherwise. The pointer can be used to reuse this memory, while a null pointer will cause the reuse to create a fresh allocation. If such a match is found but not all control flow paths consume the reuse token, the token is dropped in those paths by the garbage collection from the third step. Counting Immutable Beans does not describe how to perform this analysis for join points. However, the original implementation in Lean does perform reuse in the context of join points, and served as an example for the previous reuse implementation in Roc. But the lean implementation is quite limited in the reuse opportunities it can find: new allocations in the body or continuation of a join point can only be matched with pattern matches before the join or in the body/continuation itself. Reset opportunities inside the continuation cannot be matches to reuse opportunities inside the body. This can hinder memory reuse, which is especially annoying since join points were supposed to be cheap, a problem we attempt to solve in section 3.4.

Another heuristic algorithm is used to infer the borrow signature of the functions. Every parameter is assumed to be borrowed to reduce reference counting operations, but can be marked as owned for one of these three reasons:

- The function calls `reset` on the parameter. Marking such a variable as borrowed could result in the reset statement returning a pointer to the allocation for later reuse, whilst the structure itself is actually shared, resulting in the mutation of structures that should have remained unchanged.

- The function is tail called from a function that can be tail call optimised where the variable is owned. Because marking the parameter as borrowed would force the caller to insert a drop after the call which could break the optimisation.

- The parameter is passed to a function that takes the given parameter as owned.

Other parameters are marked as borrowed to reduce the number of inserted reference count operations. The reference counting analysis uses a borrowed and an owned context with variables. Dups are inserted to move a variable from the borrowed to the owned context, before each consumption (like constructors or owned function parameters), which moves them back to the borrowed context. Drops are inserted after the latest usage of an owned variable if not already consumed.

For example:

```
Tuple a b : {l: a, r: b}

toTuple : Str -> Tuple Str Str
toTuple = \x -> {l: x, r: x}
```

Listing 7: Ownership example

In the Roc example above the function `toTuple` is determined to take its argument as borrowed (as it does not call `reset` on it or a function that does) and the value is duped once because there now is an additional reference to the value, which looks roughly like this in the intermediate representation (IR) of Roc:

```
toTuple : Str -> Tuple Str Str
toTuple = \x ->
    dup x;
    {l: x, r: x}
```

Listing 8: Ownership example

But if we pattern match on a reference counted value and do not use that value again, but do construct a value of the same type again, a `reset` and `reuse` pair will get inserted.

For example:

```
LinkedList a : [
    Cons a (LinkedList a),
    Nil
]

swapHead : LinkedList a, a ->
    LinkedList a
swapHead = \list, x ->
    when list is
        Nil -> Nil
        Cons _ ys -> Cons x ys
```

Listing 9: swapHead

In the example above the reference counted list is scrutinised and not used again in the branches. Thus the memory of the `Cons` can be reused. Next, reference counting analysis makes sure to drop the now unused `x` and `list` in the `Nil` branch and dup the `ys` in the `Cons` branch. Note that `x` is marked as owned as it is used inside a constructor.

```
swapHead : LinkedList a, a ->
    LinkedList a
swapHead = \list, x ->
    when list is
        Nil ->
            drop x
            drop list
            Nil
        Cons _ ys ->
            dup ys
            r = reset list
            Cons@r x ys
```

Listing 10: IR swapHead

2.2.3 Perceus

In contrast to Counting Immutable Beans (before inserting reference count operations), which analyses functions in order to determine a suitable ownership for their parameters, Perceus from Reinking et al opts to always pass parameters as owned. Owning all parameters can result in an increase in reference count operations. But the authors of Perceus argue that being able to instantly free unused memory, which is not possible while borrowing said memory, increases cache locality and decreases memory usage. A program in which all memory is deallocated as soon as it becomes garbage is called "garbage free".

This difference in garbage memory is noticeable when a function takes a parameter as borrowed while it is called with an owned variable, which results in the value staying in memory throughout the execution of the called function. Even when it is used for a fraction of this time. For example:

```
fib : Nat -> Nat
fib = \len ->
    list = List.repeat len (len * 1000000)
    fibHelp list

fibHelp : List Nat -> Nat
fibHelp = \list ->
    len = (List.len list) // 1000000
    when len is
        0 -> 0
        1 -> 1
        _ -> fib (len - 1) +
             fib (len - 2)
```

Listing 11: Borrowing Owned Variables

The function `fib` allocates a list with the same length as its input, and passes it to `fibHelp` which converts it back to a number to determine the next step. Thus, the list is marked as borrowed when using Counting Immutable Beans. Meaning that `fib` will look like the following:

```
fib = \len ->
    list = List.repeat len (len * 1000000)
    result = fibHelp list
    drop list
    result
```

Listing 12: Fib IR

Dropping the list after the function finishes means that the entire `list` will remain allocated during the entire remaining calculation of `fibHelp`, including the recursive calls. Passing the list, or any parameter, as owned, allows for their immediate deallocation when no longer in use. We realise that this particular example is somewhat contrived, but every borrowed parameter could have the same effect once called with an owned variable. We will look at another downside of borrowing in section 2.4.1.

2.3 Reuse Analysis

The goal of reuse analysis is to find matching deallocations and allocations, and to *reuse* the already allocated memory instead of deallocating it and allocating new memory. In practice this is done by inserting *reset* statements which take a variable, and returns a pointer to its memory if the reference count of that variable is unique, or a null pointer otherwise. After which a *reuse* statement can use this pointer to reuse the memory for a new structure, or allocate new memory in case of a null pointer, as described in subsection 2.2.2. Note that reusing memory by definition results in a program that

is no longer garbage free, since reuse holds onto garbage memory. It is desirable to insert the least number of reset/reuse operations possible, while maintaining a certain level of actual reuse. Meaning that reset statements on always non-unique variables should be avoided. Because their logic is more expensive than a simple decrement while providing no benefit. Additionally, the latest available deallocation should be reused. So that earlier memory can be deallocated as soon as possible. To reduce the overall program memory footprint. Roc lowlevel functions, those which are defined in the compiler itself using Zig, can perform some basis memory reuse as well. The `listSortWith` function for example can simply sort the list in place if the list RC is unique. A welcome benefit from reference counting, but not something we will focus on in the following subsections.

2.3.1 Drop Guided Reuse

Both *Counting Immutable Beans* and *Perceus* describe reuse analysis as a pass performed *before* reference counting, which allows the reference counting algorithm to cleanup unused reuse tokens and (in the case of the former) prevents borrowing from breaking reuse opportunities. However, Lorenzen and Leijen explain in their paper *Reference Counting with Frame Limited Reuse* [2] that this approach has drawbacks whenever a reset opportunity (a pattern match) appears before the latest usage of that variable. If the reset statement is inserted *before* the latest usage, reference counting will insert a dup right before the reset. Resulting in the variable never being unique during reset and always failing reuse. We can demonstrate this problem using the following function:

```
modifyHead : LinkedList a,
    (LinkedList a -> a) -> LinkedList a
modifyHead = \list, f ->
    when list is
        Cons x xs ->
            y = f list
            Cons y xs
        Nil -> Nil
```

`modifyHead` takes a linked list and, in the case of a non-empty list, uses a function that takes the entire list (as owned) to determine a new value for the head. The reuse analysis will recognise the pattern match on `list` and will try to insert a reuse for the allocation thereafter. Reuse, after reference counting, will have the following result:

```
modifyHead = \list, f ->
    when list is
        Cons x xs ->
            dup xs
            dup list
            r = reset list
            y = f list
            Cons @r y xs
        Nil -> Nil
```

Where we can see the `reset list` will always return null because the list will never be unique.

If a reset statement in inserted *after* the latest usage of a variable in a function call, the algorithm will insert a dup right before the function call. And then the function will make sure it's dropped again. Allowing the subsequent reset/reuse to effectively reuse the memory. But this approach means that the function itself is never able to reuse the memory, because the reference count is always greater than one, and that the entire structure (including members) is kept in memory during the entire execution of the (potentially recursively) called function. Keeping the entire structure in memory can potentially result in an arbitrary increase in peak memory usage. If we look at our previous example:

```
modifyHead = \list, f ->
    when list is
        Cons x xs ->
            dup xs
            dup list
            y = f list
            r = reset list
            Cons @r y xs
        Nil -> Nil
```

We can now see that the reuse might work, if the list was unique to begin with. But the entire list (including its children) will stay allocated during the execution of `f`, in addition to the function not being able to reuse the list itself. `f` not being able to reuse the list does not seem like a problem in this small example. But such behaviour could result in an unbounded increase of allocated memory for recursive functions with larger allocated structures.

The alternative presented by Lorenzen and Leijen is *Drop Guided Reuse*. A reuse analysis pass that is performed *after* reference counting. This algorithm (as the name suggests) tries to replace already inserted drops with resets. The downside of this approach is that we explicitly have to free unused reuse tokens. But this is relatively straightforward. The upside is, how-

ever, that reuse is only performed when a variable would already have been dropped according to the (in the case of Perceus) garbage free RC algorithm. Thus, resets after dups are avoided (because a drop won't follow a dup). And most importantly, a reset will only keep hold of the memory for a single structure. Meaning that the increase in peak memory usage is bounded by a constant (as each stack frame can only have so many resets) multiplied by the number of stack frames, *Frame Limited Reuse*. This results in our example looking like this:

```
modifyHead = \list, f ->
    when list is
        Cons x xs ->
            dup xs
            y = f list
            Cons y xs
        Nil -> Nil
```

Where reference counting does not insert any drops, and drop specialisation subsequently does not insert any resets. Meaning no unbounded amount of garbage memory is being hold on to, while `f` can reuse the `list` or drop it as soon as it is no longer in use.

2.4 Specialisation

In addition to the reference counting algorithm itself, *Perceus: Garbage Free Reference Counting with Reuse* describes two (ignoring reuse analysis) additional optimisations that can be performed to improve performance: *drop specialisation* and *reuse specialisation*.

2.4.1 Drop Specialisation

Drops behave, in the case of general structures, like the following pseudo-code:

```
drop = \structure ->
    if unique structure
        then
            child1 = structure.1
            drop child1
            ...
            childn = structure.n
            drop childn
            free structure
        else
            decref structure
```

Listing 13: Drop Naive

If the reference count of the structure is unique, all children are dropped recursively after which the original structure is freed. If it's not unique, the reference count is simply decremented by one using *decref*.

A pattern frequently observed, however, is dups of children being inserted before the drop of a parent. Like in a map function:

```
mapLinkedList :
    LinkedList a, (a -> b)
    -> LinkedList b
mapLinkedList = \l, f -> when l is
    Cons x xs ->
        dup x
        dup xs
        drop l
        Cons (f x) (mapLinkedList xs f)
    Nil -> Nil
```

Listing 14: mapLinkedList naive

That results in the children being projected out inside the drop while they were already in scope in the original function. And the children being duped right before being dropped again whenever the parent is unique. This is the exact problem drop specialisation tries to remedy. It does so by inlining drops (whenever children are duped before, or are already projected out) and moving these dups inside the uniqueness branches to cancel out, which would look like this using join points:

```
mapLinkedList = \l, f -> when l is
    Cons x xs ->
        join continuation =
            Cons (f x) (mapLinkedList xs f)
        if unique structure
            then
                free l
                jump continuation
            else
                dup x
                dup xs
                decref l
                jump continuation
    Nil -> Nil
```

Listing 15: mapLinkedList specialised

And as expected: all reference counting operations for the children have been removed from the unique path. Note that this technique is not only relevant for unions but boxes (a heap allocated value), records and lists of a known size as well, albeit in a modified shape.

Drop specialisation works with Counting Immutable Beans as well, borrowed references are never dropped by their functions and as such won't be specialised. Thus their reference count won't be checked and they won't be freed prematurely. However, in addition to the borrowing

function not being able to specialise the drop, the caller itself is unlikely to be able to do so as well, as it doesn't have access to (duped) children.

2.4.2 Drop Guided Drop Specialisation

The Perceus paper mentioned drop specialisation as an optimisation technique, but did not specify any specific algorithm for finding matching dups and drops to specialise. Koka itself performs drop specialisation as an optional step during reference counting. The algorithm generates a list of variables to be dupped and a list of variables to be dropped for each branch of a pattern match. Corresponding dups and drops are subsequently removed or drop specialised. This works for simple cases like mapping over linked lists, but fails to specialise drops across *multiple* pattern matches and drops *without* any pattern matches (like record indexing). Let us consider the `fst` function again:

```
fst : Tuple a b -> a
fst = \t -> t.l
```

Structs themselves are not reference counted, but can be duped/dropped which will dup/drop any fields that *are* reference counted. The left item will be dupped right before the tuple will be dropped. Resulting in the following IR:

```
fst = \t ->
    l = t.l
    dup l
    drop t
    l
```

That dupes `l` only for it to be dropped again during the drop of `t`. Similar problems occur when a scrutinee is borrowed after pattern matching. As that will cause the drop to be inserted later in the analysis.

Our novel *drop guided drop specialisation* takes a different approach and is performed in a separate pass. We formalise our algorithm in section 2.4.1 but will outline the general idea here:

- Encountered dups are added to a "duped" environment.
- Pattern matches and projection add the parent/child relation to a "child" environment.
- Encountered drops check the duped environment for the dropped variable, in which case the drop is removed. Or for children of the dropped variable, in which case the drop is specialised using these children.
- Dups consumed by later drops are removed from the function.
- The duped environment is (partially) cleared when function calls or other drops are encountered. As removing dups beforehand might cause values to be freed prematurely.

This approach gives the following IR for our previous example:

```
fst = \t ->
    l = t.l
    r = t.l
    drop r
    l
```

where the reference count of `l` is not modified and just the other item is dropped.

Drop guided drop specialisation can be performed after reference counting and either before or after reuse analysis. (in the case of drop guided reuse, which is performed after reference counting). But performing it before reuse analysis has three few key benefits: Firstly, when drop specialisation is performed before reuse analysis, no reset/reuse operations have been inserted in the IR yet, simplifying analysis. Secondly, lowlevel functions that borrow their arguments (like projections) can cause patterns like these to be emitted `dup x; .... drop x;`. These patterns cause *drop* guided reuse to reuse the dropped `x`. But since it was incremented before, the reuse will always fail. And as the drop is now replaced by a reuse it is more difficult for drop specialisation to match/remove the `dup` and `drop` together where possible. Thirdly, drop specialisation inserts branches on the uniqueness of variables. These non-unique branches (where the reuse of the variable is certainly impossible) allow the reuse analysis to not consider the final decref of a variable to be a reuse opportunity. Instead, an older and potentially non-null reuse token can be passed. For example:

```
swapTail : LinkedList a, LinkedList a
    -> LinkedList a
swapTail = \l, r -> when l is
    Nil -> Nil
    Cons x xs -> when r is
        Nil -> Nil
        Cons y ys -> Cons y xs
```

Listing 16: Double Reuse

The `swapTail` function adds the head of the second list to the tail of the first list, if both are non-empty lists. Without drop specialisation reuse analysis would find a drop for `r` in the

last branch, inserted by reference counting analysis, and try to reuse it. But if `r` does not have a unique reference count, the reuse fails. Drop specialisation however, in an attempt to reduce reference counting, branches on the uniqueness of `r`. Creating a path from the drop of `l` to the construction of a new `Cons` where no other reuse opportunities exist. Resulting in an IR similar to this:

```
swapTail = \l, r -> when l is
    Nil ->
        drop r
        Nil
    Cons x xs ->
        join jp1 r1 =
            when r is
                Nil ->
                    free r1
                    drop xs
                Cons y ys ->
                    join jp2 r2 =
                        Cons @r2 y xs
                    if unique r
                        then
                            free r1
                            drop ys
                            r2 = reset r
                            jump jp2 r2
                        else
                            decref r
                            jump jp2 r1

        if unique l
            then
                r1 = reset l
                jump jp1 r1
            else
                dup xs
                decref r
                jump jp1 null
```

Listing 17: Double Reuse IR

Where we can see that an unique `l` will be used for reuse in the case where a non-unique `r` won't be. Additionally, the unique path can be optimised as well. Instead of calling `reset` on the pointer which will return the pointer if its reference count is unique, we can simply pass along the pointer directly as reuse token since we *know* it's unique. These effects are something we will see back in the result section as well. Since Koka performs drop specialisation during reference counting, they too perform specialisation before reuse analysis. But they do not seem to use known (non-)unique reference counts to improve their reuse analysis.

2.4.3 Reuse Specialisation

Reuse specialisation is an optimisation that aims to only update the fields that actually change between the original value and the new value in the when using reuse. Instead of writing over every field on construction. This analysis should be compatible with the work from this paper, but having multiple potential reuse tokens for a single allocation does limit the reuse specialisation opportunities. The optimisation is not implemented in Roc and as such no adaptations have been made in this paper.

3 Formalisation

We formalise our results using a normalized linear resource calculus λ^{1n}. An extended version of the normalized calculus as described in Reference Counting with Frame Limited Reuse [2]. Which in turn is a normalized version of the calculus as described in Perceus: Garbage Free Reference Counting with Reuse [1], an untyped lambda calculus with bindings and mutually exclusive pattern matches. The normalisation of a program means to translate a program with expressions as arguments to functions and constructors, to one where the expressions are replaced by variables that are bound in newly generated bindings [6], see section 2.1.3. For example, the expression $e_1\ e_2$ (application) will be translated into let $x = e_2$; $e_1\ x$. Normalisation does not only allow us to be more explicit in regards to the programs evaluation order, but creates a syntax that better corresponds with the intermediate representation of Roc programs during compilation as well. In this section we will use a combination of lists, sets, and multisets to define our evaluation context and operations. We will use the compact comma notation to for set manipulation. (S, x) denotes x being added to (multi)set S. And (S, T) denotes the combination of the combination of two (multi) sets.

3.1 Calculus

The syntax of λ^{1n} is shown in figure 1.

$e ::= v$	(value)
$\mid e\ x$	(application)
$\mid \text{let } x = e;\ e$	(bind)
$\mid \text{match } x\ \{p_i \hat{\rightarrow} e_i\}$	(match)
$\mid \text{dup } x;\ e$	(dup)
$\mid \text{drop } x;\ e$	(drop)
$\mid \text{decref } x;\ e$	(decref)
$\mid \text{free } x;\ e$	(free)
$\mid \text{let } r = \text{reset } x\ ;\ e$	(reset)
$\mid \text{letp } y = \text{project } N\ x\ ;\ e$	(projection)
$\mid \text{join id } \bar{\text{r}} = \lambda x.e;\ e$	(joinpoint)
$\mid \text{jump id } x\ \bar{\text{r}}$	(jump)
$v ::= x$	(variable)
$\mid \lambda x.e$	(function)
$\mid \text{C } \bar{x}$	(constructor)
$\mid \text{C@}r\ \bar{x}$	(reuse)
$\mid \text{null}$	(null pointer)
$p ::= \text{C } \bar{b}$	(pattern)
$b ::= x$	(binder)
$\mid \_$	(wildcard)

Figure 1: Normalized Linear resource calculus syntax of λ^{1n}

The constructs highlighted in gray are those that can only be inserted by the compiler and those highlighted in blue are new additions to the calculus. These additions are used to perform drop specialisation and to accommodate join points and projections as used by the Roc compiler, which require a non-trivial adaptation to RC and reuse analyses. Later figures will have new additions highlighted in blue as well.

The effect that each statement has on the heap (their semantics) is described in the next section.

3.2 Heap Semantics

In figure 2 we show the heap semantics of the defined calculus. Here we show the effect every statement has on the heap H. The heap is defined by a mapping from variables to their value together with a reference count. In addition, we keep track of all in-scope joinpoints using mapping J.

Every rule $\frac{condition}{\text{H}|\text{J}\vdash e \rightsquigarrow \text{H}|\text{J}\vdash e'}$ can be read as follows: Given a heap H and a joinpoint environment J where the *condition* holds. Then the expression e evaluates to (derives) e' together with an updated heap and joinpoint environment. For example; the [DROP$_h$] rule is applied when:

- The current heap contains the variable x with a reference count higher than one (the reference is not unique).
- The expression being evaluated is drop x; e.

And it evaluates to the same heap with a decremented reference count with now expression e being next to evaluate. The evaluation of a small example expression can be seen in listing 18. Note that the $\square$ (the "hole") is used to indicate which expression is currently being evaluated [20].

```
Assuming bound variables a and b
(not shown on the heap),
and a remaining expression e

Initial state
{} | {} ⊢
let x = C a b;  match x {C d e → drop x; e}

Evaluating constructor
let x = □;  match x {C d e → drop x; e}
{} | {} ⊢ C a b

Evaluated constructor ([CON_h])
let x = □;  match x {C d e → drop x; e}
{z →¹ C a b} | {} ⊢ z

Evaluating binding
{z →¹ C a b} | {} ⊢
let x = z;  match x {C d e → drop x; e}

Evaluated binding ([LET_h])
{z →¹ C a b} | {} ⊢
 match z {C d e → drop z; e}

Evaluated match ([MATCH_h])
{z →¹ C a b} | {} ⊢ drop z; e

Evaluated drop ([DCON_h])
{} | {} ⊢ drop a; drop b; e
```

Listing 18: Evaluation

Later formalizations will use a similar syntax, albeit with different environments.

$\mathrm{H} : x \to^{\mathbb{N}^+} v$ maps variables to their values with a reference count
$\mathrm{J} : id \to (\bar{\mathrm{r}}, \lambda x.e)$ keeps track of the available joinpoints and their reuse tokens
$E ::= \Box \mid E\ x \mid \text{let } x = E;\ e \mid$
Eval: $\frac{\mathrm{H}\mid\mathrm{J}\vdash e \rightsquigarrow \mathrm{H}'\mid\mathrm{J}'\vdash e'}{\mathrm{H}\mid\mathrm{J}\vdash E[e] \rightsquigarrow \mathrm{H}'\mid\mathrm{J}'\vdash E[e']}$

$$\frac{\text{fresh } f}{\mathrm{H} \mid \mathrm{J} \vdash \lambda^\Gamma x.e \rightsquigarrow \mathrm{H}, f \to^1 \lambda^\Gamma x.e \mid \mathrm{J} \vdash f}\ [\mathrm{LAM}_h]$$

$$\frac{\text{fresh } z}{\mathrm{H} \mid \mathrm{J} \vdash \mathrm{C}\ \bar{x} \rightsquigarrow \mathrm{H}, z \to^1 \mathrm{C}\ \bar{x} \mid \mathrm{J} \vdash z}\ [\mathrm{CON}_h]$$

$$\frac{(f \to^n \lambda^\Gamma x.e) \in \mathrm{H}}{\mathrm{H} \mid \mathrm{J} \vdash f\ y \rightsquigarrow \mathrm{H} \mid \mathrm{J} \vdash \text{dup } \Gamma;\ \text{drop } f;\ e[x := y]}\ [\mathrm{APP}_h]$$

$$\frac{p_i = \mathrm{C}\ \bar{x} \quad (x \to^n \mathrm{C}\ \bar{z}) \in \mathrm{H}}{\mathrm{H} \mid \mathrm{J} \vdash\ \text{match } x\ \{p_i \to e_i\}\ \rightsquigarrow \mathrm{H} \mid \mathrm{J} \vdash e_i[\bar{x} := \bar{z}]}\ [\mathrm{MATCH}_h]$$

$$\frac{}{\mathrm{H} \mid \mathrm{J} \vdash \text{let } x = z;\ e \rightsquigarrow \mathrm{H} \mid \mathrm{J} \vdash e[x := z]}\ [\mathrm{LET}_h]$$

$$\frac{}{\mathrm{H}, x \to^n v \mid \mathrm{J} \vdash \text{dup } x;\ e \rightsquigarrow \mathrm{H}, x \to^{n+1} v \mid \mathrm{J} \vdash e}\ [\mathrm{DUP}_h]$$

$$\frac{}{\mathrm{H}, x \to^{n+1} v \mid \mathrm{J} \vdash \text{drop } x;\ e \rightsquigarrow \mathrm{H}, x \to^n v \mid \mathrm{J} \vdash e}\ [\mathrm{DROP}_h]$$

$$\frac{}{\mathrm{H}, x \to^1 \lambda^\Gamma y.e_2 \mid \mathrm{J} \vdash \text{drop } x;\ e \rightsquigarrow \mathrm{H} \mid \mathrm{J} \vdash \text{drop } \Gamma;\ e}\ [\mathrm{DLAM}_h]$$

$$\frac{}{\mathrm{H}, x \to^1 \mathrm{C}\ \bar{y} \mid \mathrm{J} \vdash \text{drop } x;\ e \rightsquigarrow \mathrm{H} \mid \mathrm{J} \vdash \text{drop } \bar{y};\ e}\ [\mathrm{DCON}_h]$$

$$\frac{}{\mathrm{H}, x \to^{n+1} v \mid \mathrm{J} \vdash \text{decref } x;\ e \rightsquigarrow \mathrm{H}, x \to^n v \mid \mathrm{J} \vdash e}\ [\mathrm{DECREF}_h]$$

$$\frac{}{\mathrm{H}, x \to^1 v \mid \mathrm{J} \vdash \text{free } x;\ e \rightsquigarrow \mathrm{H} \mid \mathrm{J} \vdash e}\ [\mathrm{FREE}_h]$$

$$\frac{x \to^n \mathrm{C}\ \bar{z} \in \mathrm{H} \quad i < |\bar{z}|}{\mathrm{H} \mid \mathrm{J} \vdash \text{letp } y = \text{project } i\ x\ ;\ e \rightsquigarrow e[y := \bar{z}_i]}\ [\mathrm{PROJECT}_h]$$

$$\frac{}{\mathrm{H} \mid \mathrm{J} \vdash\ \text{join } id\ \bar{\mathrm{r}} = \lambda x.e_1;\ e_2 \rightsquigarrow \mathrm{H} \mid \mathrm{J}, id \to (\bar{\mathrm{r}}, \lambda x.e_1) \vdash e_2}\ [\mathrm{JOIN}_h]$$

$$\frac{(id \to (\bar{\mathrm{r}}, \lambda x.e)) \in \mathrm{J}}{\mathrm{H} \mid \mathrm{J} \vdash\ \text{jump } id\ y\ \bar{\mathrm{r}}'\ \rightsquigarrow \mathrm{H} \mid \mathrm{J} \vdash e[x := y, \bar{\mathrm{r}} := \bar{\mathrm{r}}']}\ [\mathrm{JUMP}_h]$$

$$\frac{\text{fresh } z}{\mathrm{H}, x \to^1 \mathrm{C}\ \bar{x} \mid \mathrm{J} \vdash \text{let } r = \text{reset } x\ ;\ e \rightsquigarrow \mathrm{H}, z \to^1 \mathrm{C}\ \bar{x} \mid \mathrm{J} \vdash \text{drop } \bar{x};\ e[r := z]}\ [\text{RESET-U}_h]$$

$$\frac{\text{fresh } z \quad n \geq 1}{\mathrm{H}, x \to^{n+1} v \mid \mathrm{J} \vdash \text{let } r = \text{reset } x\ ;\ e \rightsquigarrow \mathrm{H}, x \to^n v, z \to^1 \text{NULL} \mid \mathrm{J} \vdash e[r := z]}\ [\mathrm{RESET}_h]$$

$$\frac{}{\mathrm{H}, \mathrm{r} \to^1 \mathrm{C}\ \bar{y} \mid \mathrm{J} \vdash \mathrm{C@r}\ \bar{x} \rightsquigarrow \mathrm{H}, \mathrm{r} \to^1 \mathrm{C}\ \bar{x} \mid \mathrm{J} \vdash \mathrm{r}}\ [\text{REUSE-U}_h]$$

$$\frac{\text{fresh } z}{\mathrm{H}, \mathrm{r} \to^1 \text{NULL} \mid \mathrm{J} \vdash \mathrm{C@r}\ \bar{x} \rightsquigarrow \mathrm{H}, z \to^1 \mathrm{C}\ \bar{x} \mid \mathrm{J} \vdash z}\ [\mathrm{REUSE}_h]$$

Figure 2: Heap semantics of λ^{1n}

The [PROJECT$_h$] rule behaves like the [MATCH$_h$] rule, but instead of an entire vector, it replaces a single variable in the following expression. Given that the matched variable points to a constructor on the heap with enough fields to index.

For a let $r =$ reset x ; e the [RESET-U$_h$] rule will be applied if the reference to x is unique, and [RESET$_h$] otherwise. In the unique case, the value assigned to r will be a pointer to the original memory from x. This pointer allows a later [REUSE-U$_h$] to *reuse* that memory instead of allocating new memory. As is the case for [RESET$_h$] and [REUSE$_h$], where a *NULL* value assigned to r communicates that x was not unique and its memory could not be reused. Note that the children of x are dropped during [RESET-U$_h$], and not during reuse. To prevent them from remaining in memory for any longer than necessary.

Rules [DUP$_h$] to [FREE$_h$] correspond to the functions from the overview section (2). Any manipulated variables passed to these functions should be alive on the heap. [DUP$_h$] simply increments the reference count for the variable. While [DROP$_h$] decrements it, if it is not unique. Otherwise either [DLAM$_h$] or [DCON$_h$] are applied to free a function or constructor and drop their closure/children. The [DECREF$_h$] rule is similar to [DROP$_h$], but it can only be used when the reference count of the variable is known to be unique. Meaning it could save on some checks during actual program execution. And finally, the [FREE$_h$] rule is used to free the memory of a unique variable *without* dropping the closure/children beforehand. Free can be used in drop specialisation to replace duping children before dropping the parent.

And finally we have the [JOIN$_h$] and [JUMP$_h$] rules. That define adding joinpoints to the environment and jumping to them with an argument and reuse tokens respectively.

3.3 Reference Counting

In figure 3 we define the declarative rules that allows us to translate an expression from λ^{1n} into one with reference counting operations. The rules use a borrowed environment Δ, an owned environment Γ and environment Θ to keep track of the current joinpoint closures. All new variables from either bindings or function parameters start off in the owned environment. After which they can be consumed exactly once, by either passing them to a function or using them in a constructor. If they are not consumed they have to be explicitly dropped. And if they need to be consumed multiple times, they need to be explicitly duped multiple times. A variable is placed in the borrowed environment whilst evaluating the expressions before their last consumption. Borrowing allows some rules like pattern matching to read their value without having to insert dups and drops. As the value is guaranteed to stay alive (and not get deallocated) due to the later usage. Note that the normalisation of applications and constructors result in a reversed evaluation order. Instead of evaluating e_1 e_2 by treating the free variables of e_2 as borrowed in e_1. The normalized version of let $x = e_2$; e_1 x uses the free variables from e_1 as borrowed in e_2 instead.

The rule [VAR$_{cd}$] allows any variable to be derived when only that variable is currently owned, which requires the [DUP$_{cd}$] and [DROP$_{cd}$] rules to be applied such that variables are used as owned exactly as many times as they are added to the owned environment. The [APP$_{cd}$] rule allows x to be borrowed during the derivation of e. Borrowing allows x to be used for matches or projections without having to increment and decrement the reference count of x. And a similar pattern can be seen in [BIND$_{cd}$] where the owned environment Γ_2, used to derive e_2', is borrowed during the derivation of e_1'. [MATCH$_{cd}$] derives new expressions for each of its branches, where the bound variables (in the pattern) are duped and added to the owned environment. And again, a similar pattern can be seen for the rule [PROJECT$_{cd}$]. The [CON$_{cd}$] rule requires all variables to be owned since they will stay referenced by the new structure. Lastly, the rule for [JOIN$_{cd}$] adds a new joinpoint with its closure to the environment so that a final [JUMP$_{cd}$] can make sure that exactly these variables are owned when jumping to the joinpoint.

For these rules to be correct, they must result in an evaluation such that all values are alive when they will still be used and that values that are no longer used eventually will be deallocated. In other words: the reference counts have to match the actual number of references. We claim (without formal proof) that these invariants are upheld by our additions to the calculus. The [PROJECT$_{cd}$] rule dups the projected variable and adds it to the owned environment and the [JOIN$_{cd}$] rule evaluates e_1 under the assumption that the variables from Θ' with x are owned. An invariant upheld by [JUMP$_{cd}$]. Applications of [DUP$_{cd}$] and [DROP$_{cd}$] to meet the rule precondition will subsequently ensure correctness.

In addition to the declarative rules from figure 3 that aim to insert RC operations *correctly*, we provide syntax-directed derivation rules for the insertion of RC operations in λ^{1n}. These rules attempts to insert these operations *optimally*. That is, inserting decrements right after the latest usage of a variable (to lower the memory usage) and inserting the minimum amount of reference counting operations while remaining

$\Delta : \{x, ...\}$ contains all variables that are currently borrowed
$\Gamma : \{x, ...\}$ contains all variables that are currently owned
$\Theta : id \to \Gamma$ maps join point ids to their closure

$$\frac{}{\Delta \mid x \mid \Theta \vdash x \rightsquigarrow x}\,[\text{VAR}_{cd}]$$

$$\frac{x \in \Delta, \Gamma \quad \Delta \mid \Gamma, x \mid \Theta \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma \mid \Theta \vdash e \rightsquigarrow \text{dup } x;\ e'}\,[\text{DUP}_{cd}]$$

$$\frac{\Delta \mid \Gamma \mid \Theta \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \mid \Theta \vdash e \rightsquigarrow \text{drop } x;\ e'}\,[\text{DROP}_{cd}]$$

$$\frac{\Delta, x \mid \Gamma \mid \emptyset \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \mid \emptyset \vdash e\ x \rightsquigarrow e'\ x}\,[\text{APP}_{cd}]$$

$$\frac{\emptyset \mid \Gamma, x \mid \emptyset \vdash e \rightsquigarrow e' \quad \Gamma = \text{fv}(\lambda x.e)}{\Delta \mid \Gamma \mid \emptyset \vdash \lambda x.e \rightsquigarrow \lambda^{\Gamma} x.e'}\,[\text{LAM}_{cd}]$$

$$\frac{x \notin \Delta, \Gamma_1, \Gamma_2 \quad \Delta, \Gamma_2 \mid \Gamma_1 \mid \emptyset \vdash e_1 \rightsquigarrow e_1' \quad \Delta \mid \Gamma_2, x \mid \Theta \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \mid \Theta \vdash \text{let } x = e_1;\ e_2 \rightsquigarrow \text{let } x = e_1';\ e_2'}\,[\text{BIND}_{cd}]$$

$$\frac{x \in \Delta, \Gamma \quad \Delta \mid \Gamma, \bar{z}_i \mid \Theta \vdash e_i \rightsquigarrow e_i' \quad \bar{z}_i = \text{bv}(p_i)}{\Delta \mid \Gamma \mid \Theta \vdash\ \text{match } x\ \{\overline{p_i \to e_i}\}\ \rightsquigarrow\ \text{match } x\ \{\overline{p_i \to \text{dup } \bar{z}_i;\ e_i'}\}}\,[\text{MATCH}_{cd}]$$

$$\frac{}{\Delta \mid \bar{x} \mid \Theta \vdash C\ \bar{x} \rightsquigarrow C\ \bar{x}}\,[\text{CON}_{cd}]$$

$$\frac{x \in \Delta, \Gamma \quad \Delta \mid \Gamma, y \mid \Theta \vdash e_i \rightsquigarrow e_i'}{\Delta \mid \Gamma \mid \Theta \vdash \text{letp } y = \text{project } i\ x\ ;\ e \rightsquigarrow \text{letp } y = \text{project } i\ x\ ;\ \text{dup } y;\ e}\,[\text{PROJECT}_{cd}]$$

$$\frac{\Gamma' = \text{fv}(\lambda x.e_1) \quad \Theta' = \Theta, id \to \Gamma' \quad \emptyset \mid \Gamma', x \mid \Theta' \vdash e_1 \rightsquigarrow e_1' \quad \Delta \mid \Gamma \mid \Theta' \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma \mid \Theta \vdash\ \text{join } id\ [] = \lambda x.e_1;\ e_2 \rightsquigarrow\ \text{join } id\ [] = \lambda x.e_1';\ e_2'}\,[\text{JOIN}_{cd}]$$

$$\frac{}{\emptyset \mid \Gamma, x \mid \Theta, id \to \Gamma \vdash\ \text{jump } id\ x\ []\ \rightsquigarrow\ \text{jump } id\ x\ []}\,[\text{JUMP}_{cd}]$$

Figure 3: Declarative linear resource rules of λ^{1n} for reference counting.

$$\frac{}{\Delta \mid x \mid \Theta \vdash x \rightsquigarrow x}\,[\text{VAR}_{cd}]$$

$$\frac{x \in \Delta, \Gamma \quad \Delta, x \mid \Gamma \mid \emptyset \vdash e \rightsquigarrow e' \quad \bar{x}' = [x] - \Gamma}{\Delta \mid \Gamma \mid \emptyset \vdash e\ x \rightsquigarrow \text{dup}\ \bar{x}';\ e'\ x}\,[\text{APP}_{cs}]$$

$$\frac{\emptyset \mid ys, x \mid \emptyset \vdash e \rightsquigarrow e' \quad ys = \text{fv}(\lambda x.e) \quad \Delta_1 = ys - \Gamma \quad \bar{x}' = [x] - \text{fv}(e)}{\Delta, \Delta_1 \mid \Gamma \mid \emptyset \vdash \lambda x.e \rightsquigarrow \text{dup}\ \Delta_1;\ \lambda^{ys} x.(\text{drop}\ \bar{x}';\ e')}\,[\text{LAM}_{cs}]$$

$$\frac{x \notin \Delta, \Gamma \quad \Gamma_2 = \Gamma \cap \text{fv}(e_2) \quad \Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \mid \emptyset \vdash e_1 \rightsquigarrow e_1' \quad \Delta \mid \Gamma_2, x \mid \Theta \vdash e_2 \rightsquigarrow e_2' \quad \bar{x}' = [x] - \text{fv}(e_2)}{\Delta \mid \Gamma \mid \Theta \vdash \text{let}\ x = e_1;\ e_2 \rightsquigarrow \text{let}\ x = e_1';\ \text{drop}\ \bar{x}';\ e_2'}\,[\text{BIND}_{cs}]$$

$$\frac{\Delta \mid \Gamma_i \mid \Theta \vdash e_i \rightsquigarrow e_i' \quad \Gamma_i = (\Gamma, \bar{z}_i) \cap \text{fv}(e_i) \quad \Gamma_i' = (\Gamma, \bar{z}_i) - \Gamma_i \quad \bar{z}_i = \text{bv}(p_i)}{\Delta \mid \Gamma, x \mid \Theta \vdash\ \text{match}\ x\ \{p_i \to e_i\}\ \rightsquigarrow\ \text{match}\ x\ \{p_i \to \text{dup}\ \bar{z}_i;\ \text{drop}\ \Gamma_i';\ e_i'\}}\,[\text{MATCH}_{cs}]$$

$$\frac{\bar{y} = \bar{x} - \Gamma \quad \{\bar{y}\} \equiv \Delta}{\Delta \mid \Gamma \mid \emptyset \vdash C\ \bar{x} \rightsquigarrow \text{dup}\ \bar{y};\ C\ \bar{x}}\,[\text{CON}_{cs}]$$

$$\frac{\Delta \mid \Gamma' \mid \Theta \vdash e_i \rightsquigarrow e_i' \quad \Gamma' = (\Gamma, y) \cap \text{fv}(e) \quad \bar{x}' = [x, y] - \text{fv}(e)}{\Delta \mid \Gamma \mid \Theta \vdash \text{letp}\ y = \text{project}\ i\ x\ ;\ e \rightsquigarrow \text{letp}\ y = \text{project}\ i\ x\ ;\ \text{dup}\ y;\ \text{drop}\ \bar{x}';\ e'}\,[\text{PROJECT}_{cs}]$$

$$\frac{\Gamma' = \text{fv}(\lambda x.e_1) \quad \bar{x}' = [x] - \text{fv}(e_1) \quad \Theta' = \Theta, id \to \Gamma' \quad \emptyset \mid \Gamma', x \mid \Theta'\ e_1 \rightsquigarrow e_1' \quad \Delta \mid \Gamma \mid \Theta'\ e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma \mid \Theta \vdash\ \text{join}\ id\ [] = \lambda x.e_1;\ e_2 \rightsquigarrow\ \text{join}\ id\ [] = \lambda x.(\text{drop}\ \bar{x}';\ e_1');\ e_2'}\,[\text{JOIN}_{cs}]$$

$$\frac{\Gamma_1 \supseteq \Gamma_2 \quad \Gamma_3 = \Gamma_1 - \Gamma_2 \quad \{x\} \equiv \Delta, \Gamma_3 \quad \bar{x}' = [x] - \Gamma_3}{\Delta \mid \Gamma_1 \mid \Theta, id \to \Gamma_2 \vdash\ \text{jump}\ id\ x\ []\ \rightsquigarrow \text{dup}\ \bar{x}';\ \ \text{jump}\ id\ x\ []}\,[\text{JUMP}_{cs}]$$

Figure 4: Syntax-directed linear resource rules of λ^{1n} for reference counting.

correct. For example: for any (in scope) variable x the declarative rules [DUP$_{cd}$] and [DROP$_{cd}$] allow any number of matching dup and drop statements to be inserted. And while translating e' to dup x; drop X; e' is *correct*, it does not result in an efficient program.
Additionally, the declarative rules are non-deterministic. Meaning that for a given expressions there might be multiple rules that are possible to apply to obtain the next expression. Non-determinism is no problem when illustrating the concept or proving correctness. But does not provide a straight forward conversion into an actual implementation. For example: during the evaluation of any expression it is possible to apply the [DUP$_{cd}$] and [DROP$_{cd}$] rules, in addition to the rule specific to the expression like [BIND$_{cd}$] or [MATCH$_{cd}$]. And for rule [BIND$_{cd}$], the owned environment Γ is split into the two environments Γ_1 and Γ_2 for the evaluation of e_1 and e_2 respectively. But exactly how to split this environment such that both expressions can be translated is not specified. The syntax-directed rules, which are largely inspired by the work for Perceus [1], are described in figure 4.

The [BIND$_{cs}$] rule now explicitly states that all owned variables that are free (defined before but used) in expression e_2, including the newly bound variable x. Should be considered owned during the evaluation to e_2', whilst being borrowed during the evaluation of expression e_1. Additionally, x itself should be dropped immediately when not used in e_2. Just like the parameters to lambdas in [LAM$_{cs}$] and [JOIN$_{cs}$].

The [MATCH$_{cs}$] rule dups the variables from the bound pattern for each branch and drops all variables that are not used in the branch. Every variable should not be dropped in at least one branch. Otherwise it should have been dropped before. The [PROJECT$_{cs}$] rule behaves similarly, potentially dropping the assigned variable and the projected variable if they are not used in the continuation. Both these rules generate a dup followed by a drop for any unused bound variables. This is something drop-specialisation solves as well.

The [JUMP$_{cs}$] rule takes the closure of the join point that is being jumped to. All of these variables being currently owned. With x being the only other variable in the current environment as either borrowed or owned. If x is borrowed (thus not in Γ_3) it will be duplicated to make it owned. This can occur when x is both a parameter and part of the closure to the same joinpoint, effectively consuming it twice. The reuse token lists for both joins and jumps are empty, because this pass is performed before reuse analysis, which adds these reuse tokens.

In addition to being correct, these rules should result in a *garbage free* evaluation. The [PROJECT$_{cs}$] rule retains garbage freeness by duping the new variable (and considering it

owned) and dropping either this variable or the projected one if it's their last usage. The [JOIN$_{cs}$] can do so by dropping its parameter if they are not used.

3.4 Reuse Analysis

In figure 5 we can see the declarative rules for drop-guided reuse [2]. The rules can be read like those for reference counting. But instead of keeping track of the owned and borrowed variables, we use:

- S to keep track of the memory layout of each variable.
- R to map each such layout to the reuse tokens available for that layout.
- J to map join points to a list of layouts that will be used to pass reuse tokens to joinpoints.

These rules will see the first usage of reset and reuse. In addition adding reuse token variables to the join and jumps.

The [VAR$_{rd}$] rule again requires the current R map to be empty. Because the reuse analysis is performed after reference counting (as oppose to before in Perceus), which means that unused reuse tokens are no longer cleaned up themselves and we are now responsible for making sure that any obtained reuse token is used (or freed). The variable layouts get inserted by the [MATCH$_{rd}$] rule for its branches after discriminating the variable and by the [BIND-CON$_{rd}$] rule, which inserts the layout of the bound constructor for the evaluation of the continuation. Knowing the layout of a variable after construction is especially useful in scenarios where a constructor is never matched on before dropping it (e.g. by projecting it after creation). The [RESET$_{rd}$] and [RESET-U$_{rd}$] rules are responsible for creating reuse tokens and inserting them in the environment. The [RESET$_{rd}$] can be applied when evaluating a drop statement while knowing the layout of the dropped variable. While the [RESET-U$_{rd}$] can be used to remove a free in order to reuse the memory later. These reuse tokens can then be consumed by either [REUSE$_{rd}$] which passes the reuse token to the constructor to allow for reuse or [FREE-R$_{rd}$] which frees the memory for an unused reuse token if the token is not null. Because the reuse analysis pass is performed after drop specialisation (section 2.4.2), which inserts both free and decref operations. We have the ability to simply add variables to the reuse environment instead of freeing them, which is better than inserting a reset, as a reset has to check whether a variable is unique at runtime. While a freed variable is known to be unique at compile time. Similarly, decrefs are only inserted when a variable is known *not* to be unique. Allowing us to avoid the insertion of a reset. Finally, [JOIN$_{rd}$] uses J to communicate the layouts of the reuse tokens it expects. Allowing later [JUMP$_{rd}$]s to pass correct tokens.

And again we define syntax-directed resource rules as well. See figure 6.

These syntax-directed rules allow us to explicitly define when it is beneficial to insert reset/reuses and when not. While it would be correct to simply always apply the [DROP$_{rs}$] rule instead of the [RESET$_{rs}$] rule. That would not be very useful.

The [BIND$_{rs}$] rule is updated to first evaluate the bound expression and pass any unused reuse tokens to the next expression. The [MATCH$_{rs}$] rule combines all the reuse tokens that are actually used in each branch, to only drop those that used in other branches but not in the same one. The rules for frees and drops are updated to only remove the free / insert a reset respectively when the reuse token is actually used in the next expression, which prevents additional cleanup if the token is never used in the first place. And we can see that the jump in [RJUMP-N$_{rs}$] is annotated with the layouts available for reuse at that point. These layouts will be used by the ml function to determine the optimal number of reuse tokens for the joinpoint. (e.g., the maximum number of reuse tokens for each layout). In the second pass over the jumps, with the reuse token layouts for the joinpoint in the context, we can determine what reuse token to pass for which layout. We do this by iterating the layouts in reverse and removing tokens from the context R$_n$ once they have been consumed. Context R$'$ is used as a backup environment, where a reuse token variable is created for each layout so that they can be assigned NULL and passed as a reuse token if no other token is available. The layouts are iterated in reverse to make sure the most recent reuse token is consumed first in the jointpoint, as would be the case without joinpoints. In addition, reversing the order makes sure that NULL reuse tokens won't be used before the tokens that might not be null. Something we will mention in section 5.2 for future work as well.

To proof the correctness of the declarative resource rules for drop-guided reuse from figure 5. We again would need to proof that values are not deallocated prematurely, and that they would eventually be deallocated when no longer in use. Including the reuse tokens. For the syntax-directed rules from figure 6 we would need to proof that all reuse opportunities are utilised, if unique.

$\mathrm{S} : x \to \mathrm{l}$ maps variables to their memory layout
$\mathrm{R} : [\mathrm{l} \to \mathrm{r}, ...]$ ordered list that maps layouts to their reuse tokens
$\mathrm{J} : id \to \bar{\mathrm{l}}$ maps join point ids to a list of layouts

$$\frac{}{\mathrm{S} \mid \emptyset \mid \mathrm{J} \vdash x \rightsquigarrow x}\,[\mathrm{VAR}_{rd}]$$

$$\frac{}{\mathrm{S} \mid \emptyset \mid \mathrm{J} \vdash C\ v_1 ... v_n \rightsquigarrow C\ v_1 ... v_n}\,[\mathrm{CON}_{rd}]$$

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \emptyset \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R} \mid \emptyset \vdash e\ x \rightsquigarrow e'\ x}\,[\mathrm{APP}_{rd}]$$

$$\frac{\emptyset \mid \emptyset \mid \emptyset \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \emptyset \mid \emptyset \vdash \lambda\ x.e \rightsquigarrow \lambda\ x.e'}\,[\mathrm{LAM}_{rd}]$$

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \mathrm{dup}\ x;\ e \rightsquigarrow \mathrm{dup}\ x;\ e'}\,[\mathrm{DUP}_{rd}]$$

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R}, \mathrm{l} \to \mathrm{r} \mid \mathrm{J} \vdash e \rightsquigarrow \mathrm{free\ r};\ e'}\,[\text{FREE-R}_{rd}]$$

$$\frac{x \to \mathrm{l} \in \mathrm{S} \quad \mathrm{S} \mid \mathrm{R}, \mathrm{l} \to \mathrm{r} \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad \text{fresh r}}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \mathrm{drop}\ x;\ e \rightsquigarrow \mathrm{let\ r} = \mathrm{reset}\ x\ ;\ e'}\,[\mathrm{RESET}_{rd}]$$

$$\frac{\mathrm{S} \mid \mathrm{R}_1 \mid \emptyset \vdash e_1 \rightsquigarrow e_1' \quad \mathrm{S} \mid \mathrm{R}_2 \mid \mathrm{J} \vdash e_2 \rightsquigarrow e_2' \quad e_1 \neq C\ v_1 ... v_n}{\mathrm{S} \mid \mathrm{R}_1, \mathrm{R}_2 \mid \mathrm{J} \vdash \mathrm{let}\ x = e_1;\ e_2 \rightsquigarrow \mathrm{let}\ x = e_1';\ e_2'}\,[\mathrm{BIND}_{rd}]$$

$$\frac{\mathrm{S}, x \to \mathrm{l} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad \mathrm{l} = \mathit{layout}(C)}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \mathrm{let}\ x = C\ v_1 ... v_n;\ e \rightsquigarrow \mathrm{let}\ x = C\ v_1 ... v_n;\ e'}\,[\text{BIND-CON}_{rd}]$$

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \mathrm{drop}\ x;\ e \rightsquigarrow \mathrm{drop}\ x;\ e'}\,[\mathrm{DROP}_{rd}]$$

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \mathrm{decref}\ x;\ e \rightsquigarrow \mathrm{decref}\ x;\ e'}\,[\mathrm{DECREF}_{rd}]$$

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \mathrm{free}\ x;\ e \rightsquigarrow \mathrm{free}\ x;\ e'}\,[\mathrm{FREE}_{rd}]$$

$$\frac{x \to \mathrm{l} \in \mathrm{S} \quad \mathrm{S} \mid \mathrm{R}, \mathrm{l} \to x \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad \text{fresh r}}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \mathrm{free}\ x;\ e \rightsquigarrow e'}\,[\text{RESET-U}_{rd}]$$

$$\frac{\mathrm{S}, x \to \mathrm{l}_i \mid \mathrm{R} \mid \mathrm{J} \vdash e_i \rightsquigarrow e_i' \quad \mathrm{l}_i = \mathit{layout}(p_i)}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash\ \mathrm{match}\ x\ \{\overline{p_i \to e_i}\}\ \rightsquigarrow\ \mathrm{match}\ x\ \{\overline{p_i \to e_i'}\}}\,[\mathrm{MATCH}_{rd}]$$

$$\frac{\mathrm{S}, y \to \mathrm{l} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad \mathrm{l} = \mathit{layout}(x_i)}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \mathrm{letp}\ y = \mathrm{project}\ i\ x\ ;\ e \rightsquigarrow \mathrm{letp}\ y = \mathrm{project}\ i\ x\ ;\ e'}\,[\mathrm{PROJECT}_{rd}]$$

$$\frac{\mathrm{l} = \mathit{layout}(C)}{\mathrm{S} \mid \mathrm{l} \to \mathrm{r} \mid \mathrm{J} \vdash \mathrm{C}\ \bar{x} \rightsquigarrow \mathrm{C@r}\ \bar{x}}\,[\mathrm{REUSE}_{rd}]$$

$$\frac{\begin{array}{c} n \in \mathbb{N} \quad \bar{\mathrm{r}} = [\mathrm{fresh\ r} \mid n' \in [0..n]] \quad \bar{\mathrm{l}} = [\mathrm{fresh\ l} \mid n' \in [0..n]] \quad \mathrm{R}_2 = \{\bar{\mathrm{l}}_{n'} \to \bar{\mathrm{r}}_{n'} \mid n' \in [0..n]\} \\ \mathrm{J}_2 = \mathrm{J}_1, id \to \bar{\mathrm{l}} \quad \emptyset \mid \mathrm{R}_2 \mid \mathrm{J}_2 \vdash e_1 \to e_1' \quad \mathrm{S} \mid \mathrm{R}_1 \mid \mathrm{J}_2 \vdash e_2 \to e_2' \end{array}}{\mathrm{S} \mid \mathrm{R}_1 \mid \mathrm{J}_1 \vdash\ \mathrm{join}\ id\ [] = \lambda x.e_1;\ e2 \to\ \mathrm{join}\ id\ \bar{\mathrm{r}} = \lambda x.e_1';\ e2'}\,[\mathrm{JOIN}_{rd}]$$

$$\frac{id \to \bar{\mathrm{l}} \in \mathrm{J} \quad \bar{\mathrm{l}} \equiv \{\mathrm{l} \mid \mathrm{l} \to \mathrm{r} \in \mathrm{R}\} \quad \bar{\mathrm{r}} = [\mathrm{r} \mid \mathrm{l} \in \bar{\mathrm{l}}, \mathrm{l} \to \mathrm{r} \in \mathrm{R}]}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash\ \mathrm{jump}\ id\ x\ []\ \to\ \mathrm{jump}\ id\ x\ \bar{\mathrm{r}}}\,[\mathrm{JUMP}_{rd}]$$

$$\frac{\text{fresh l} \quad \text{fresh r} \quad \mathrm{S} \mid \mathrm{R}, \mathrm{l} \to \mathrm{r} \mid \mathrm{J} \vdash e \to e'}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash e \to \mathrm{let\ r} = \mathrm{NULL};\ e'}\,[\mathrm{NULL}_{rd}]$$

Figure 5: Declarative linear resource rules of λ^{1n} for drop-guided reuse.

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \emptyset \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R} \mid \emptyset \vdash e\ x \rightsquigarrow e'\ x} \quad [\mathrm{APP}_{rs}]$$

$$\frac{\emptyset \mid [] \mid \emptyset \vdash e \rightsquigarrow e'}{\mathrm{S} \mid [] \mid \emptyset \vdash \lambda x.e \rightsquigarrow \lambda x.e'} \quad [\mathrm{LAM}_{rs}]$$

$$\frac{\mathrm{S} \mid \mathrm{R}_1 \mid \emptyset \vdash e_1 \rightsquigarrow e_1' \quad \mathrm{S} \mid \mathrm{R}_2 \mid \mathrm{J} \vdash e_2 \rightsquigarrow e_2' \quad \mathrm{R}_2 = [\mathrm{l} \to \mathrm{r} \mid \mathrm{l} \to \mathrm{r} \in \mathrm{R}_1, \mathrm{r} \notin \mathrm{fv}(e_1')] \quad e_1 \neq C\ v_1 ... v_n}{\mathrm{S} \mid \mathrm{R}_1 \mid \mathrm{J} \vdash \text{let } x = e_1;\ e_2 \rightsquigarrow \text{let } x = e_1';\ e_2'} \quad [\mathrm{BIND}_{rs}]$$

$$\frac{\mathrm{S}, x \to \mathrm{l} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad \mathrm{l} = \mathit{layout}(C)}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{let } x = C\ v_1 ... v_n;\ e \rightsquigarrow \text{let } x = C\ v_1 ... v_n;\ e'} \quad [\text{BIND-CON}_{rs}]$$

$$\frac{\mathrm{l}_i = \mathit{layout}(p_i) \quad \mathrm{S}, x \to \mathrm{l}_i \mid \mathrm{R} \mid \mathrm{J} \vdash e_i \rightsquigarrow e_i' \quad \bar{\mathrm{r}} = \{\mathrm{r} \mid \mathrm{l} \to \mathrm{r} \in \mathrm{R}, \mathrm{r} \in (\mathrm{fv}(e_1') \cup ... \cup \mathrm{fv}(e_n')\} \quad \bar{\mathrm{r}}_i = \bar{\mathrm{r}} - \mathrm{fv}(e_i')}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{match } x\ \{p_i \to e_i\} \rightsquigarrow \text{match } x\ \{p_i \to \text{free } \bar{\mathrm{r}}_i;\ e_i'\}} \quad [\mathrm{MATCH}_{rs}]$$

$$\frac{\mathrm{S}, y \to \mathrm{l} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad \mathrm{l} = \mathit{layout}(x_i)}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{letp } y = \text{project } i\ x\ ;\ e \rightsquigarrow \text{letp } y = \text{project } i\ x\ ;\ e'} \quad [\mathrm{PROJECT}_{rs}]$$

$$\frac{\mathrm{l} = \mathit{layout}(C)}{\mathrm{S} \mid \mathrm{R}, \mathrm{l} \to \mathrm{r} \mid \emptyset \vdash \mathrm{C}\ \bar{x} \rightsquigarrow \mathrm{C@r}\ \bar{x}} \quad [\mathrm{REUSE}_{rs}]$$

$$\frac{\mathrm{l} = \mathit{layout}(C) \quad \mathrm{l} \to \mathrm{r} \notin \mathrm{R}}{\mathrm{S} \mid \mathrm{R} \mid \emptyset \vdash C\ v_1 ... v_n \rightsquigarrow C\ v_1 ... v_n} \quad [\mathrm{CON}_{rs}]$$

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{dup } x;\ e \rightsquigarrow \text{dup } x;\ e'} \quad [\mathrm{DUP}_{rs}]$$

$$\frac{x \to \mathrm{l} \in \mathrm{S} \quad \text{fresh r} \quad \mathrm{S} \mid \mathrm{R}, \mathrm{l} \to \mathrm{r} \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad \mathrm{r} \in \mathrm{fv}(e')}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{drop } x;\ e \rightsquigarrow \text{let r} = \text{reset } x\ ;\ e'} \quad [\mathrm{RESET}_{rs}]$$

$$\frac{x \to \mathrm{l} \in \mathrm{S} \quad \text{fresh r} \quad \mathrm{S} \mid \mathrm{R}, \mathrm{l} \to \mathrm{r} \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad \mathrm{r} \notin \mathrm{fv}(e')}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{drop } x;\ e \rightsquigarrow \text{drop } x;\ e'} \quad [\mathrm{DROP}_{rs}]$$

$$\frac{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash e \rightsquigarrow e'}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{decref } x;\ e \rightsquigarrow \text{decref } x;\ e'} \quad [\mathrm{DECREF}_{rs}]$$

$$\frac{x \to \mathrm{l} \in \mathrm{S} \quad \mathrm{S} \mid \mathrm{R}, \mathrm{l} \to x \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad x \in \mathrm{fv}(e')}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{free } x;\ e \rightsquigarrow e'} \quad [\text{RESET-U}_{rs}]$$

$$\frac{x \to \mathrm{l} \in \mathrm{S} \quad \mathrm{S} \mid \mathrm{R}, \mathrm{l} \to x \mid \mathrm{J} \vdash e \rightsquigarrow e' \quad x \notin \mathrm{fv}(e')}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{free } x;\ e \rightsquigarrow \text{free } x;\ e'} \quad [\mathrm{FREE}_{rs}]$$

$$\frac{\begin{array}{c}\mathrm{S} \mid \mathrm{R}_1 \mid \mathrm{J} \vdash e_2 \rightsquigarrow e_2' \qquad \mathrm{R}_2 = [\mathrm{l} \to \mathrm{r} \mid \mathrm{l} \in \mathrm{ml}(\mathrm{id}, e_2'), \text{fresh r}] \\ \emptyset \mid \mathrm{R}_2 \mid \emptyset \vdash e_1 \rightsquigarrow e_1' \quad \mathrm{R}_3 = [\mathrm{l} \to \mathrm{r} \mid \mathrm{l} \to \mathrm{r} \in \mathrm{R}_2, \mathrm{r} \in \mathrm{bv}(e_1')] \quad \bar{\mathrm{l}} = [\mathrm{l} \mid \mathrm{l} \to \mathrm{r} \in \mathrm{R}_3] \\ \emptyset \mid \mathrm{R}_3 \mid \mathrm{id} \to \bar{\mathrm{l}} \vdash e_1 \rightsquigarrow e_1'' \quad \mathrm{S} \mid \mathrm{R}_1 \mid \mathrm{J}, \mathrm{id} \to \bar{\mathrm{l}} \vdash e_2 \rightsquigarrow e_2'' \quad \bar{\mathrm{r}} = [\mathrm{r} \mid \mathrm{l} \to \mathrm{r} \in \mathrm{R}_3]\end{array}}{\mathrm{S} \mid \mathrm{R}_1 \mid \mathrm{J} \vdash \text{join id } [] = \lambda x.e_1;\ e_2 \rightsquigarrow \text{join id } \bar{\mathrm{r}} = \lambda x.e_1'';\ e_2''} \quad [\mathrm{JOIN}_{rs}]$$

$$\frac{id \to \bar{\mathrm{l}} \notin \mathrm{J} \quad \bar{\mathrm{l}} = [\mathrm{l} \mid \mathrm{l} \to \mathrm{r} \in \mathrm{R}]}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{jump } id\ x\ [] \rightsquigarrow \text{jump}^{\bar{\mathrm{l}}}\ id\ x\ []} \quad [\text{JUMP-N}_{rs}]$$

$$\frac{\begin{array}{c}id \to \bar{\mathrm{l}} \in \mathrm{J} \quad \mathrm{R}' = [\mathrm{l} \to \text{fresh r} \mid \mathrm{l} \in \bar{\mathrm{l}}] \\ \mathrm{R}_{|\bar{\mathrm{l}}|} = \mathrm{R} \quad rs = [\mathrm{r}, \mathrm{R}_{n-1} = \mathrm{R}_n - [\mathrm{l} \to \mathrm{r}] \mid n \in [\mid \bar{\mathrm{l}} \mid ..0], \mathrm{l} = \bar{\mathrm{l}}_n, \mathrm{l} \to \mathrm{r} \in \mathrm{R}_n \cup \mathrm{R}'] \\ \bar{\mathrm{r}} = [\mathrm{r} \mid \_ \to \mathrm{r} \in \mathrm{R}', \mathrm{r} \in rs]\end{array}}{\mathrm{S} \mid \mathrm{R} \mid \mathrm{J} \vdash \text{jump } id\ x\ [] \rightsquigarrow \text{let } \bar{\mathrm{r}}_0 = \text{NULL};\ \text{let } \bar{\mathrm{r}}_n = \text{NULL};\ \text{jump } id\ x\ rs} \quad [\mathrm{JUMP}_{rs}]$$

Figure 6: Syntax-directed linear resource rules of λ^{1n} for drop-guided reuse.

P : $x \to (y, i)$ maps variables to projected children and at which index D : $\{x, ...\}$ is a multi-set of variables, indicating the number of times each variable has been duped so far.

3.5 Drop Guided Drop Specialisation

In figure 7 we show the formalised rules for the syntax directed drop specialisation. These rules, in addition to the evaluated expression, return the modified environment D. Not unlike the rules from the heap semantics from figure 1. This environment contains the variables that have been duped (multiple times) before. We need to return this updated environment D in order to determine whether an increment was used in a later drop specialisation or not. Because if it was not used, the increment should remain. And because the variable can still be used in other language constructs like function calls or constructors without affecting this environment, we can no longer rely on the free variables from an expression to determine if a variable was used or not. Like we did in figure 3 to 6. Application of these rules should not cause reference counts to be non-zero where originally zero and zero where originally not zero to be correct. To prevent premature or overdue deallocation.

The [MATCH$_{ds}$], [PROJECT$_{ds}$], and [BIND-CON$_{ds}$] rules add parent child relations to the environment P. The [PROJECT$_{ds}$] rule can simply add the new child relation given the index, and return the updated expression together with its updated environment. The [MATCH$_{ds}$] rule, however, is a bit more involved. For each branch it first adds the projections to the environment using the bound variables from the pattern (not the wildcards) and evaluates the branch expression using this updated environment. Then, for each unique variable in the original environment D, the minimum multiplicity (the frequency of an item in a multi-set) of that variable in every branch is used to create a new multi-set D$'$. The difference in the multiplicity for each variable between this new multi-set and the one returned by the evaluation of each branch is used to add additional dups in the respective branch. To compensate for the usage of (dups of) the same variable in drop specialisation in other branches. The [BIND-CON$_{ds}$] rule adds the parent child relations for the new constructor in a similar way as the [MATCH$_{ds}$] rule did. This rule allows a decrement of the newly constructed value to be drop specialised, even if its fields are not projected or pattern matched yet.
The [DUP$_{ds}$] and [DUP-FIRST$_{ds}$] rules are used to insert variables into the D environment. Note that the first occurrence of a *DUP* of a variable x will always be evaluated using rule [DUP-FIRST$_{ds}$] while later occurrences are evaluated using [DUP$_{ds}$]. This allows us the [DUP$_{ds}$] rule to propagate unused dups upwards towards the [DUP-FIRST$_{ds}$], where all dups can be combined into one and removed from the environment. Combining dups can save on additional reference counting operations when the IR allows a dup to increment the reference count by more than one.
The [DROP-REMOVE$_{ds}$], [DROP-SPECIALISE$_{ds}$], and [DROP$_{ds}$] rules can then be used to consume these dups. The [DROP-REMOVE$_{ds}$] is used to remove matching dups and subsequent drops from the expression entirely. A pattern frequently observed when projecting out a field just to read its value, resulting in a dup for the projection and a drop after reading its value. The [DROP-SPECIALISE$_{ds}$] performs the actual drop specialisation. Drop specialisation is used when at least one of the fields from the variable is already in scope as a variable. If so, all the duped children of x are placed in D$'$. So that they can be removed during the evaluation of the expression to prevent an increment to be moved after the decrement of its parent (potentially freeing both while the child is still in use). And given to the *specialise* which can consume some of them and add back the remainder to the return environment. The *specialise* function uses the reference count uniqueness of x to specialise the drop it by branching: if x is unique, all fields except those that are in the duped child environment are dropped. Including the children from P$'$ which had not been projected out, but will be in this branch. After which x is freed. If x is not unique, only those that are in the duped environment are duped after which x is decreffed (lowering the reference count without having to check for uniqueness to drop children, because x is not unique in this branch). See example 19:

```
stringHeadOrEmpty :
    LinkedList Str -> Str
stringHeadOrEmpty =
    \list -> when list is
        Cons x _ ->
            dup x
            drop list
            x
        Nil ->
            drop list
            ""
```

Listing 19: Specialisation

where the 0 index for `list` to `x` is inserted in environment P and the following `dup` of `x` is inserted into D. After which the [DROP-SPECIALISE$_{ds}$] rule is applied. In the unique case the index 1 is projected out and dropped, because it was discarded using a wildcard before. After which `list` is freed. In the

$$\frac{\mathrm{P} \mid \mathrm{D} \vdash e \rightsquigarrow \mathrm{D}' \vdash e'}{\mathrm{P} \mid \mathrm{D} \vdash e\ x \rightsquigarrow \mathrm{D}' \vdash e'\ x} \quad [\mathrm{APP}_{ds}]$$

$$\frac{\mathrm{P} \mid \emptyset \vdash e \rightsquigarrow \mathrm{D}' \vdash e'}{\mathrm{P} \mid \mathrm{D} \vdash \lambda x.e \rightsquigarrow \mathrm{D} \vdash \lambda x.e'} \quad [\mathrm{LAM}_{ds}]$$

$$\frac{\mathrm{P} \mid \mathrm{D} \vdash e_1 \rightsquigarrow \mathrm{D}' \vdash e_1' \quad \mathrm{P} \mid \mathrm{D}' \vdash e_2 \rightsquigarrow \mathrm{D}'' \vdash e_2' \quad e_1 \neq C\ v_1 ... v_n}{\mathrm{P} \mid \mathrm{D} \vdash \text{let } x = e_1;\ e_2 \rightsquigarrow \mathrm{D}'' \vdash \text{let } x = e_1';\ e_2'} \quad [\mathrm{BIND}_{ds}]$$

$$\frac{\mathrm{P}' \mid \mathrm{D} \vdash e \rightsquigarrow \mathrm{D}' \vdash e' \quad \mathrm{P}' = \mathrm{P} \cup \{x \to (v_i, i) \mid v_i \in v_1 ... v_n\}}{\mathrm{P} \mid \mathrm{D} \vdash \text{let } x = C\ v_1 ... v_n;\ e \rightsquigarrow \mathrm{D}' \vdash \text{let } x = C\ v_1 ... v_n;\ e'} \quad [\mathrm{BIND\text{-}CON}_{ds}]$$

$$\frac{\begin{array}{c}\mathrm{P}_i \mid \mathrm{D} \vdash e_i \rightsquigarrow \mathrm{D}_i' \vdash e_i' \quad \mathrm{P}_i = \mathrm{P} \cup \{x \to (x_i, n) \mid n \in 0..|\bar{b_i}| - 1,\ x_i = \bar{b_i}_n\} \quad \mathrm{C}\ \bar{b_i} = p_i \\ \mathrm{D}' = \bigcup \{x^m \mid\ x \in \mathrm{supp}\,\mathrm{D},\ m = \min(\mathrm{D}_1'(x), ..., \mathrm{D}_n'(x))\}\end{array}}{\mathrm{P} \mid \mathrm{D} \vdash\ \text{match } x\ \{p_i \to e_i\}\ \rightsquigarrow \mathrm{D}' \vdash\ \text{match } x\ \{p_i \to \text{dup } \mathrm{D}_i' - \mathrm{D}';\ e_i'\}} \quad [\mathrm{MATCH}_{ds}]$$

$$\frac{\mathrm{P}, x \to (y, i) \mid \mathrm{D} \vdash e \rightsquigarrow \mathrm{D}' \vdash e'}{\mathrm{P} \mid \mathrm{D} \vdash \text{letp } y = \text{project } i\ x\ ;\ e \rightsquigarrow \mathrm{D}' \vdash \text{letp } y = \text{project } i\ x\ ;\ e'} \quad [\mathrm{PROJECT}_{ds}]$$

$$\frac{\begin{array}{c}\mathrm{P} \mid \mathrm{D}, x \vdash e \rightsquigarrow \mathrm{D} \vdash e' \quad x \notin \mathrm{D} \\ n = \mathrm{D}'(x) \quad xs = x^n\end{array}}{\mathrm{P} \mid \mathrm{D}' \vdash \text{dup } x;\ e \rightsquigarrow (\mathrm{D}'' - xs) \vdash \text{dup } xs;\ e'} \quad [\mathrm{DUP\text{-}FIRST}_{ds}]$$

$$\frac{\mathrm{P} \mid \mathrm{D}, x \vdash e \rightsquigarrow \mathrm{D}' \vdash e' \qquad x \in \mathrm{D}}{\mathrm{P} \mid \mathrm{D} \vdash \text{dup } x;\ e \rightsquigarrow \mathrm{D}' \vdash e'} \quad [\mathrm{DUP}_{ds}]$$

$$\frac{\mathrm{P} \mid \mathrm{D} \vdash e \rightsquigarrow \mathrm{D}' \vdash e'}{\mathrm{P} \mid \mathrm{D}, x \vdash \text{drop } x;\ e \rightsquigarrow \mathrm{D}' \vdash e'} \quad [\mathrm{DROP\text{-}REMOVE}_{ds}]$$

$$\frac{\begin{array}{c}x \to \_ \in \mathrm{P} \quad x \notin \mathrm{D} \quad C v_1 ... v_n = layout(x) \\ \mathrm{P}' = \{x \to (\text{fresh } y, i) \mid i \in 1...n, x \to (\_, i) \notin \mathrm{P}\} \\ \mathrm{D}' = \mathrm{D} \cap \{y \mid x \to (y, i) \in \mathrm{P}\} \quad \mathrm{P}'' \mid \mathrm{D} - \mathrm{D}' \vdash e \rightsquigarrow \mathrm{D}'' \vdash e' \\ (e'', \mathrm{D}''') = \text{specialise}(\mathrm{P}, \mathrm{P}', \mathrm{D}', e')\end{array}}{\mathrm{P} \mid \mathrm{D} \vdash \text{drop } x;\ e \rightsquigarrow \mathrm{D}'' + \mathrm{D}''' \vdash e''} \quad [\mathrm{DROP\text{-}SPECIALISE}_{ds}]$$

$$\frac{\mathrm{P} \mid \mathrm{D} \vdash e \rightsquigarrow \mathrm{D}' \vdash e' \quad x \notin \mathrm{D} \quad x \to \_ \notin \mathrm{P}}{\mathrm{P} \mid \mathrm{D} \vdash \text{drop } x;\ e \rightsquigarrow \mathrm{D}' \vdash \text{drop } x;\ e'} \quad [\mathrm{DROP}_{ds}]$$

$$\frac{\mathrm{P} \mid \emptyset \vdash e_1 \rightsquigarrow \mathrm{D}' \vdash e_1' \quad \mathrm{P} \mid \mathrm{D} \vdash e_2 \rightsquigarrow \mathrm{D}'' \vdash e_2'}{\mathrm{P} \mid \mathrm{D} \vdash\ \text{join id } [] = \lambda x.e_1;\ e_2 \rightsquigarrow \mathrm{D}'' \vdash\ \text{join id } [] = \lambda x.e_1';\ e_2'} \quad [\mathrm{JOIN}_{ds}]$$

$$\frac{}{\mathrm{P} \mid \mathrm{D} \vdash\ \text{jump } id\ x\ []\ \rightsquigarrow \mathrm{D} \vdash\ \text{jump } id\ x\ []} \quad [\mathrm{JUMP}_{ds}]$$

Figure 7: Syntax-directed linear resource rules of λ^{1n} for drop specialization.

non-unique case the `x` has to be duped after which `list` can be decreffed. Resulting in an IR similar to this for the `Cons` branch:

```
if unique list
    then
        xs = project 1 list
        decref xs
        free list
        x
    else
        dup x
        decref list
        x
```

In the previous example there was only a single statement following the specialisation, the return of `x`, which is simply added to both branches. Larger continuations would instead be compiled using e.g. join points to prevent having to copy the expression into both arms.

Lastly, the [DROP$_{ds}$] rule is used to keep the drop when it cannot be removed or specialised. Note that the join point functions are evaluated without considering the jumps to it for brevity. The analysis itself is relatively straightforward:

- Both the join point body and continuation get evaluated as normal.
- The jumps store which dups they have available
- Matches take the intersection of these dups to get those available in all branches
- If the continuation has jumps to the join point (with dups) and the body has no jumps to itself, we can evaluate both the body and continuation again, assuming the join body to consume the dups from the continuation.

This final optimisation can only be performed for non-recursive join points, the number of dups returned from the body would otherwise not be monotonically non-increasing. Making it difficult to analyse.

For these rules, we assume the layout for each variable is known during evaluation, we have shown how to obtain them during reuse analysis.

4 Results

We used five benchmark programs to compare the previous Counting Immutable Beans RC algorithm with the newer Perceus based algorithm and to measure the effects of drop specialisation. We ported the original algorithm over after initially replacing it with Perceus, such that the effects of other changes to the compiler made in the meantime were limited as much as possible. The five benchmark programs originate from the Perceus paper. Most of which stem from the Lean repository [10] and the Koka repository [9]. Using these benchmarks allows us to compare the performance of Roc against Koka and Haskell as well. The benchmarks stress memory allocation and are not computationally heavy [1].

- Deriv: Calculates the derivative of expressions up to 10 million nodes and counts the resulting number of nodes.
- NQueens: Solves the nqueens problem for a 13 by 13 board and returns a list of all solutions.
- CFold: Builds an binary expression tree of 20 layers and folds all constant values.
- RBTree: Generates a red-black tree with a 4,200,000 boolean values and counts the `true` values afterwards.
- RBTreeCK: Similar to RBTree but every 10th tree is added to a list. Resulting in non-unique subtrees and the slow path with new allocations to be taken.

In addition to comparing the runtime performance with these benchmarks, we compare the frequency of reference counting operations statically and dynamically in Roc as well. This information should provide us with a better understanding of the observed changes in performance. The source for the (Roc) benchmarks can be found in our fork from the Roc repository [8].

We want to note that the original implementation did not allow for the reuse across different data structures with the same layout, but the new implementation does. The benchmarks however do not have these opportunities, and as such should not bias the results.

The computer that ran these benchmarks was a MSI GS66 Stealth 11 EU [15] with the following specifications:

- 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
- 16 GB Memory @ 3200MHz
- Micron 3400 1TB (MTFDKBA1T0TFH) M.2 2280 PCIe Gen4 SSD

The tests were performed using Ubuntu 22.04.2 LTS over WSL with 12 GB of memory and 4 cores appointed. With the main system running Windows 11 Home build 22621.1702. These specifications should have little effect on the inserted operations, except any change in IR due to the current platform, but do affect the performance of the benchmarks.

4.1 Operations

We will compare both the static (the number of inserted dups, drops (including decrefs), resets, and reuses in the IR) and dynamic (number of encountered of dups, drops, and (de)allocations during execution) program information for these benchmarks. Individual rows were colour coded to increase readability. Green cells indicate the lowest (best) value in the row and red cells indicate the highest (worst) value in the row. Values in between got a colour in between. The rows for reset and reuse were not coded, more or less reset or reuse statements do not necessarily result in better results. As this highly depends on their placement and reference counting operations. Finally, keep in mind that drop specialisation duplicates an inserted drop during specialisation. One for both uniqueness branches. Resulting in relatively more static drops than dynamic ones. And that drop specialisation allows for the reuse of memory without the insertion of resets.

From table 1 we can read that either with or without drop specialisation the number of inserted dup operations increases from Counting Immutable Beans to Perceus, while the opposite is true (albeit to a lesser extend) for the drops. While the drop specialisation pass reduces both significantly more (up to 75% for Deriv!). The reduced number of drop operations subsequently lowers the number of reset operations inserted by frame limited reuse.

From table 2 we can see a similar pattern where drop specialisation reduces the performed RC operations significantly for Beans and Perceus. And the number of allocations using Perceus seems to be lower across the board. This is a result of more valid reset/reuse opportunities due to join points being analysed for reuse as well, while this is not the case for Beans. We can also see that drop specialisation for Perceus causes a big decrease in allocations, more than 50% for RBTreeCk. This change is a result of our drop specialisation allowing reuse analysis to use multiple reset sources for a single reuse, as we explained in section 2.4.2.

4.2 Performance

To compare the benefits/drawbacks for each algorithm implementation in Roc, we have a look at both the memory usage and the time performance for each benchmark. In addition, to put these results into perspective, we compare the RC implementations for Roc with the same benchmarks in Koka and Haskell. Haskell is a popular, garbage collected, lazy, and functional programming language. We used GHC version 9.2.8 to compile the benchmarks with the `-O` flag passed to optimise the resulting binary. The benchmarks for Haskell contained strictness annotations to ensure the same amount of work was performed. For Koka, the first language to use Perceus and frame limited reuse, we used version 2.4.0 with the `-o=2` flag passed for full optimisation. The original benchmarks written for Koka contained borrow annotations to tell the compiler that a certain argument to a function should be treated as borrowed, potentially reducing the number of required RC operations. But since Roc does not have such annotations, we opted to remove them for fairness. The four benchmarks for Roc itself were compiled using the `--optimize` flag. Using the version from the aforementioned commit as is. Together with a C platform and the mimalloc memory allocator, the allocator used by both Koka and Lean [14]. To obtain the time benchmark results, all benchmark + language combinations were run a 100 times in succession using hyperfine. The results of which can be seen in figure 8.

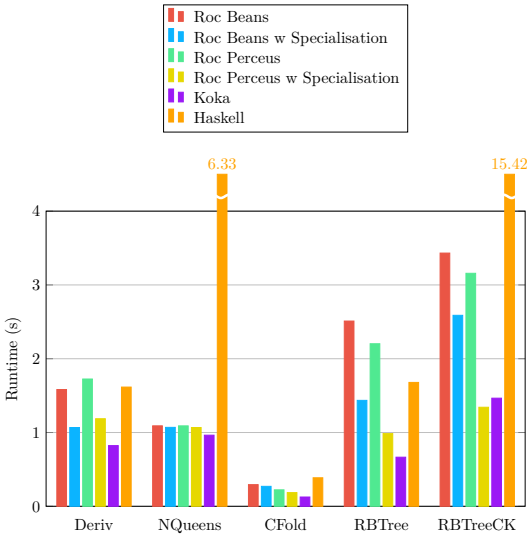

Figure 8: Benchmarks Time Performance

These results seem to match those from section 4.1 in that drop specialisation has a big positive impact on the performance for both Counting Immutable Beans and Perceus, up to 50% for Perceus. But they also show that the increase in reference counting operations due to a lack

Static		Beans	Beans Specialized	Perceus	Perceus Specialized
Deriv	dup	117	24	134	40
	drop	116	54	122	65
	reset	9	9	28	13
	reuse	10	10	22	19
NQueens	dup	8	7	10	8
	drop	10	7	14	10
	reset	0	0	0	0
	reuse	0	0	0	0
CFold	dup	54	23	61	25
	drop	65	40	48	36
	reset	6	6	18	8
	reuse	6	6	14	14
RBTree	dup	43	30	47	29
	drop	31	12	25	22
	reset	6	6	21	4
	reuse	6	6	23	17
RBTreeCk	dup	51	34	51	33
	drop	36	25	31	36
	reset	8	8	21	6
	reuse	8	8	25	25

Table 1: Benchmarks Static Operations

Dynamic		Beans	Beans Specialized	Perceus	Perceus Specialized
Deriv	dup	2.20E+08	2.53E+07	3.11E+08	1.17E+08
	drop	2.61E+08	7.11E+07	3.46E+08	1.55E+08
	alloc	3.88E+07	3.88E+07	3.44E+07	3.14E+07
	dealloc	3.88E+07	3.88E+07	3.44E+07	3.14E+07
NQueens	dup	3.07E+08	2.97E+08	3.07E+08	2.97E+08
	drop	3.16E+08	3.02E+08	3.16E+08	3.02E+08
	alloc	9.35E+06	9.35E+06	9.35E+06	9.35E+06
	dealloc	9.35E+06	9.35E+06	9.35E+06	9.35E+06
CFold	dup	2.37E+07	1.56E+07	3.48E+07	2.10E+06
	drop	3.80E+07	2.92E+07	3.90E+07	3.69E+06
	alloc	1.43E+07	1.43E+07	4.19E+06	3.15E+06
	dealloc	1.43E+07	1.43E+07	4.19E+06	3.15E+06
RBTree	dup	4.95E+08	7.98E+07	5.82E+08	9.03E+07
	drop	6.34E+08	8.40E+07	6.32E+08	1.32E+08
	alloc	1.39E+08	1.39E+08	5.04E+07	4.62E+07
	dealloc	1.39E+08	1.39E+08	5.04E+07	4.62E+07
RBTreeCk	dup	5.36E+08	2.98E+08	6.22E+08	1.39E+08
	drop	6.84E+08	3.28E+08	6.99E+08	1.65E+08
	alloc	1.48E+08	1.48E+08	7.64E+07	3.02E+07
	dealloc	1.48E+08	1.48E+08	7.64E+07	3.02E+07

Table 2: Benchmarks Dynamic Operations

of borrowing can have a measurable effect on performance, as is the case for the Deriv benchmark. Note that Roc in general performs somewhat worse than Koka, but better than Haskell. We expect this to be a result of lacking reuse specialisation and other optimisations.

To compare the memory usage, we used the `time` command to determine the "Maximum resident set size" (the maximum memory used by a program) for each combination. The results of which can be seen in figure 9.

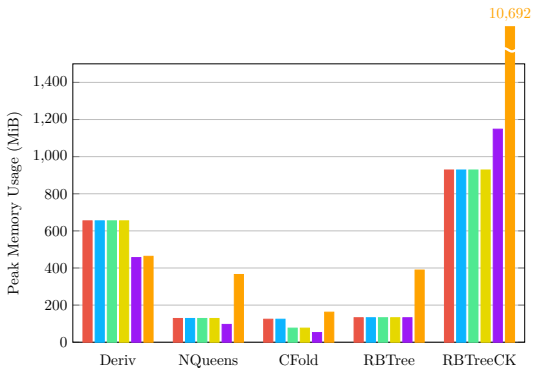

Figure 9: Benchmarks Memory Performance

Which shows that the new implementation has a similar maximum memory consumption as the old one. Except for CFold, where a slight improvement can be seen. The resemblance between the different implementations can have different causes:

- Borrowing smaller structures and deallocating them later has a smaller impact on memory usage.
- Borrowing parameters while they are actually used later in the caller has no effect on memory usage at all.
- The difference in memory usage due to borrowing might have been overshadowed by later allocations. Something we cannot determine using the maximum memory usage alone.

Additionally, it could explain why the RBTreeCK benchmark under-performed for Haskell, as Haskell had to allocate more memory. Presumably due to the inability to reuse the already allocated memory. We explained in section 2.2.3 that borrowing parameters can result in an increased memory usage. We demonstrate this using the same example from listing 11 with an input size of 20. Our new Perceus based implementation used a maximum of $172MiB$ (a single list of 20M U64's) while the Counting Immutable Beans implementation used $1878MiB$, a tenfold increase.

5 Conclusion

The resource calculus syntax used to describe both Perceus and drop-guided reuse had to be extended with join points and projections to be more compatible with the Roc compiler. As well as the additional decref and free operations as inserted by drop specialisation. In addition, both the declarative and the syntax-directed resource rules had to be modified to include rules on how to work with this extended syntax.

Perceus combined with Frame Limited Reuse itself results in an overall increase of runtime reference counting operations but a decrease in (de)allocations and memory usage when compared to Counting Immutable Beans. Performing drop specialisation after inserting the reference counting operations results in a decrease in RC operations for both algorithms. But after Perceus it has a greater impact, resulting in the least RC operations, even when compared to Counting Immutable Beans with drop specialisation. In addition, drop specialisation improves the effectiveness of reuse analysis by allowing a different reuse token to be used in cases where the original structure is known to be not unique.

5.1 Discussion

Our conclusion is based on our Roc benchmark results, but different languages that have made different choices in e.g. memory layout or allocator might see different results. Even within Roc it is possible that a part of the change in performance is a result of a change in implementation details not linked to the theoretical algorithm and rules. The current benchmarks (as any benchmark) are quite limited in the cases that get tested. Cases what would have a greater effect on non-frame-limited reuse algorithms are currently not tested (section 3.4). Different benchmarks could have led to a different conclusion. And the results for Counting Immutable Beans vs Perceus heavily rely on the cost of reference counting vs allocating new memory. Atomic reference counting (section 2.2.1) might tip the scales in favour of beans. While larger allocations might tip the scales in favour of Perceus.

5.2 Future Work

During the work for this thesis we found a few topics that need additional investigation, but were out of scope for this paper:

- This paper leans towards the practical side of reference counting, which is why we provide the syntax-directed resource rules to guide any implementation. But, without formal proofs we cannot verify that our rules do not cause operations to be inserted incorrectly. And although we believe the general concept is correct, proofs would be a welcome addition.

- The drop specialisation support for join points and jumps is currently limited to non-recursive cases. We suspect that this is an intrinsic limitation, knowing that recursive join points are a rare occurrence. But a better solution that allows drops to be specialised across join points might exist.

- Drop specialisation currently is not recursive. Meaning that any not before duped variables (which would remain dropped in the unique branch of the specialisation) are not specialised. We expect additional passes to yield relatively lower performance improvements (at the cost of an increased IR size). But there is only one way to verify these expectations.

- Not all execution flows within a function hit the same number of reuse (constructors, not drops) opportunities. Meaning that it's possible for an IR similar to this to be generated:

```
assuming r1 and r2 to be in scope
x1 = Cons @r2 1 Nil
if condition
    then
        free r1
        x1
    else
        x2 = Cons @r1 2 x1
        x2
```

  And this seems fine at first sight. But $r1$ might come from a (reused) free. Meaning that it's guaranteed not to be null, while $r2$ might be null. Using $r2$ for the first constructor would be sub-optimal for the then branch. Since it would free a non-null reuse token while allocating memory because of a null reuse token. A more advanced reuse analysis could consume reuse tokens likely to not be null first.

- We did not implement reuse specialisation for Roc, as multiple reuse sources make it difficult to determine from which variable the reuse token originates and what reuse token order for join points would result in optimal specialisation. But we do believe an implementation is possible, and it would be interesting to see the performance impact on the tested implementations, and if it has a greater impact on any of them.

- Borrowing is a tricky subject. Callees borrowing structures owned by the caller result in an increase in peak memory usage and borrowed variables cannot be used for reuse, meaning that both are mutually exclusive. Creating different borrow signatures for each function, depending on their usage, could solve some problems but results in an explosion in code size. But at the same time, borrowing structures at the right time could save on a lot of reference counting operations, as we saw for the Deriv benchmark. We think all of these problems can be solved by keeping track of ownership at *runtime*. Pointer tagging could be used to add the ownership of a pointer to a heap allocated value, which can be set by operations inserted during build time to mark pointers as borrowed for all function calls except their last one (as their last usage should be owned). Projections and pattern matches should give the projected child the same ownership, whilst creating new tag unions or records should use an owned variant of the pointer. Subsequently, all reference counting operations can check the ownership of the pointer (which is fast) and simply do nothing for borrowed variables (or always return null for reset). This approach allows all variables to be borrowed but still be reused when owned, doesn't increase the code size substantially, never causes garbage memory to remain allocated, and has minimal runtime overhead when compared to build time borrowing analysis. Roc uses pointer tagging to store the tag id for tag unions, if they fit in the pointer. Which leaves little space for the ownership bit in the case of larger tag unions. An alternative to pointer tagging might be to communicate the same information using registers.

References

[1] L.d Alex Reinking. "Perceus: Garbage Free Reference Counting with Reuse". en. In: *PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Association for Computing MachineryNew YorkNYUnited States*. 2021. DOI: 10.1145/3453483.3454032.

[2] D.L. Anton Lorenzen. "Reference Counting with Frame Limited Reuse". en. In: *Proceedings of the ACM on Programming Languages* 6 (2022), pp. 357–380. DOI: 10.1145/3547634.

[3] T.H. Axford. "Reference Counting of Cyclic Graphs for Functional Programs". en. In: *The Computer Journal* 33.5 (1990), pp. 466–470. DOI: 10.1093/comjnl/33.5.466.

[4] Henry G. Baker. "List Processing in Real Time on a Serial Computer". In: *Commun. ACM* 21.4 (Apr. 1978), pp. 280–294. ISSN: 0001-0782. DOI: 10.1145/359460.359470. URL: https://doi.org/10.1145/359460.359470.

[5] William Brandon et al. "Better Defunctionalization through Lambda Set Specialization". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591260. URL: https://doi.org/10.1145/3591260.

[6] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *SIGPLAN Not.* 28.6 (June 1993), pp. 237–247. ISSN: 0362-1340. DOI: 10.1145/173262.155113. URL: https://doi.org/10.1145/173262.155113.

[7] T.S. Jiho Choi. "Biased Reference Counting: Minimizing Atomic Operations in Garbage Collection". en. In: *PACT '18: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pp. 1–12. DOI: 10.1145/3243176.3243195.

[8] *Fork Roc GitHub*. May 2023. URL: https://github.com/JTeeuwissen/roc/tree/Porting (visited on 07/28/2023).

[9] *Koka GitHub*. June 2023. URL: https://github.com/koka-lang/koka (visited on 06/04/2023).

[10] *Lean*. June 2023. URL: https://github.com/leanprover/lean4 (visited on 06/04/2023).

[11] Henry Lieberman and Carl Hewitt. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". In: *Commun. ACM* 26.6 (June 1983), pp. 419–429. ISSN: 0001-0782. DOI: 10.1145/358141.358147. URL: https://doi.org/10.1145/358141.358147.

[12] P.D. Luke Maurer. "Compiling without Continuations". en. In: *ACM SIGPLAN Notices* 52.6 (2016), pp. 482–494. DOI: 10.1145/3140587.3062380.

[13] Simon Marlow et al. "Parallel Generational-Copying Garbage Collection with a Block-Structured Heap". In: *Proceedings of the 7th International Symposium on Memory Management*. ISMM '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 11–20. ISBN: 9781605581347. DOI: 10.1145/1375634.1375637. URL: https://doi.org/10.1145/1375634.1375637.

[14] *mimalloc GitHub*. June 2023. URL: https://github.com/microsoft/mimalloc (visited on 07/10/2023).

[15] *MSI GS66 Stealth*. 2023. URL: https://www.msi.com/Laptop/GS66-Stealth-11UX/Specification.

[16] *Roc FAQ*. May 2023. URL: https://github.com/roc-lang/roc/blob/main/FAQ.md (visited on 05/13/2023).

[17] *Roc GitHub*. May 2023. URL: https://github.com/roc-lang/roc (visited on 05/13/2023).

[18] L.d Sebastian Ullrich. "Counting Immutable Beans". en. In: *IFL '19: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. 2019, pp. 1–12. DOI: 10.1145/3412932.3412935.

[19] David Ungar, David Grove, and Hubertus Franke. "Dynamic Atomicity: Optimizing Swift Memory Management". In: *SIGPLAN Not.* 52.11 (Oct. 2017), pp. 15–26. ISSN: 0362-1340. DOI: 10.1145/3170472.3133843. URL: https://doi.org/10.1145/3170472.3133843.

[20] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1 (1994), pp. 38–94. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1093. URL: https://www.sciencedirect.com/science/article/pii/S0890540184710935.