PERCEUS FOR OCAML

A Dissertation
Presented to
The Academic Faculty

By

Elton Pinto

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science
School of Computer Science

Georgia Institute of Technology

May 2023

PERCEUS FOR OCAML

Thesis committee:

Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Daan Leijen
Principal Researcher
Microsoft Research

Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Date approved: April 28, 2023

Dedicated to my parents and my sister

ACKNOWLEDGMENTS

This work was done in collaboration with Daan Leijen and I am immensely grateful to him for giving me the opportunity to work on this project, even though it fell outside the scope of his responsibilites at Microsoft Research. Daan played a crucial role in enhancing the system's performance by providing valuable assistance in implementing the code generation of the reference counting primitives. Thank you Daan for taking the time out of your busy schedule to mentor me through the various phases of the project. This project would not have been possible without your guidance and support.

I also want to thank Prof. Vivek Sarkar for hiring me and agreeing to supervise me, especially since this project was outside his current line of work.

Thank you to my parents for all your support. I would not be where I am today without you. I also want to thank my sister Evonne Pinto for always being there for me.

Finally, I would like to thank Georgia Tech for fostering a supportive student community and productive research environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The Perceus algorithm, described in Reinking *et al.* [1], introduces a new approach for achieving automated reference counting with impressive performance capabilities. However, a direct comparison of this algorithm to a garbage collector within the same system has not been conducted, leading to an unresolved research question about whether Perceus can compete with state-of-the-art garbage collectors. In this thesis, I take a step towards answering this question by presenting an implementation of Perceus in OCaml, an industrial-strength programming language with a heavily fine-tuned and performant garbage collector. The implementation, as evaluated against the benchmark suite used in the Perceus paper, is competitive with the OCaml (version 4.14.0) garbage collector, motivating further exploration of this topic.

CHAPTER 1
INTRODUCTION

Reference counting is one of the simplest memory management schemes. In this scheme, a reference count is maintained for each object (in the loose sense), freeing it as soon as the count drops to zero. Several programming languages have employed reference counting for automated memory management — Python, Swift, Perl, and PHP, to name a few. Other languages (like C++ and Rust) support memory management through reference counting using smart pointers.

There are several benefits to using reference counting over other memory management schemes (like tracing garbage collection or manual management). For one, it is simple to implement and has a lower memory overhead. Further, reference counting does not suffer from "stop-the-world" pauses common in tracing garbage collectors as memory is managed in an incremental fashion. Finally, the reference counts can be used to enable other runtime optimizations like in-place updates and reuse.

However, there are some issues that make reference counting expensive in practice, especially in the automated setting. Maintaining reference counts can have non-trivial overhead, sometimes even eclipsing the runtime of the program's actual computation. This cost magnifies in concurrent settings, because atomic instructions need to be used, which are much more expensive than their regular counterparts. Reference counting systems are susceptible to cycles, which thereby cause memory leaks. Finally, reference counting systems can be imprecise, holding on to memory for much longer than needed when compared to manually managed memory. Owing to these issues, tracing collectors have become the more popular choice for automated memory management.

In 2021, however, Reinking *et al.* [1] shed new light on the subject by presenting Perceus: a garbage-free reference counting scheme with reuse. Unlike traditional reference

counting techniques, Perceus emits precise reference counting instructions such that the program is *garbage free* — memory is not held onto for any longer than needed. Perceus enables further optimization passes like drop specialization and reuse specialization, which enables a new programming paradigm called functional but in-place (FBIP). This paradigm allows programmers to write in-place mutating algorithms in a functional way. While Perceus does not address the performance and memory issues related to concurrency and cycles (it does mitigate it to a large extent however), Perceus innovates in regards to performance and memory usage of single-threaded cycle free programs. The authors show that Perceus, as implemented in Koka ([2]) is competitive with state-of-the memory collectors like that of OCaml, Haskell, Java, and Swift.

However, the paper compared the performance across different languages and systems and did not directly compare Perceus against a garbage collector within the same system. As such, it is an open research question if Perceus can actually be competitive with a state-of-the-art garbage collector.

In this thesis, I take a step towards answering this question by presenting an implementation of Perceus in OCaml, a widely used programming language with a rich research background and excellent performance characteristics. We can now directly compare the new Perceus reference counting background to the regular backend, which uses a finely tuned multi-generational garbage collector. The implementation, as evaluated against the benchmark suite used in the Perceus paper, is competitive with the OCaml garbage collector. This result is fascinating: with just a few months of implementation work, Perceus is able to achieve similar performance (over our memory intensive but still limited benchmark suite) as the finely turned OCaml (version 4.14.0) garbage collector.

1.1 Contributions

I offer the following contributions:

- Chapter 2 provides a short tour of OCaml, covering only concepts relevant to under-

standing our implementation of Perceus

- The OCaml compiler has been under development for over 25 years. As such, it can be daunting to understand its inner workings. Chapter 3 provides an overview of the OCaml native code compiler (as of version 4.14.0), extrapolating particularly on object representation, the middle end, and the x86 code generator. My hope is that this chapter will provide readers with the background needed to understand and appreciate our design decisions.

- Chapter 4 goes over memory management techniques, culminating with an exposition of Perceus

- Chapter 5 talks about the implementation, focusing on modifications made to the compiler and the impact of various design decisions.

- Chapter 6 presents an evaluation of the implementation. Notably, it shows that Perceus is competitive with OCaml on the selected benchmarks.

- Finally, I discuss the limitations of the implementation and directions for future work in Chapter 7

1.2 Broader Impact

This work takes a major step towards demonstrating the competitiveness of Perceus in an industrial-strength setting. The results demonstrate that reference counting, contrary to popular belief, can be a performant memory management technique. The hope is that these preliminary results can incentivize further research into reference counting, including but not limited to addressing the issues with Perceus with regards to reference cycles and concurrency.

CHAPTER 2

A TOUR OF OCAML

This chapter provides an overview of basic OCaml constructs relevant to understanding the rest of the thesis. It is geared towards readers unfamiliar with the language.

2.1 Basic expressions

Expressions are the basic unit of computation. OCaml has a number of built-in primitives for creating expressions.

Here is an expression on integers:

```
3 + 4 * 7 ;;
```

And here is an expression on floats:

```
(2.0 *. 3.14) +. 5.0 ;;
```

Two things to note:

- OCaml is *strictly typed* and will not accept ill-typed programs. For example, the expression `3 + 4.0;;` will be rejected by the compiler because + operates on integers (and 4.0 is a float).
- The double semi-colon (`;;`) is a directive that tells the compiler to evaluate everything leading up to it.

  OCaml supports conditional expressions using `if ... then ... else`.

```
let x = true in
if x then
  1 + 1
else
```

```
  2 + 2
;;
```

2.2 Let expressions

You can name expressions by using a *let binding*.

```
let x = 3 + 4 * 7 in x ;;
```

In this example, the name of the bound expression is called a *variable*. Variables in OCaml are not mutable storage locations and should not be conflated with those available in imperative programming languages like C or Python. They are simply constructs for naming sub-expressions.

Let bindings can be nested.

```
let x = 3 + 4 * 7 in
let y = 2 + 5 in
x + y ;;
```

2.3 Functions

The `let` keyword can also be used for defining functions.

```
let double x = x * 2
let volume_of_cylinder r h = 2 *. 3.14 *. r *. h ;;
```

Recursive functions must be marked as such using the `rec` keyword.

```
let rec fibonacci n =
  if n <= 0 then
    0
  else
    fibonacci (n-1) + fibonacci (n-2)
```

```
;;
```

You can define mutually recursive functions using the `and` keyword.

```
let rec even n =
  if n = 0 then true
  else odd (n - 1)
and odd n =
  if n = 0 then true
  else even (n - 1)
;;
```

OCaml also has support for anonymous (i.e. unnamed) functions (thereby making functions *first-class*).

```
(function x -> x + 1) ;;
```

2.4 Types

2.4.1 Type inference

Notice that despite OCaml being strictly typed, we did not have to specify types in any of the example programs. This feature is known as *type inference*.

Type inference is not perfect and can fail at times. In such cases, the programmer can explicitly specify types.

2.4.2 Built-in data types

In addition to integers, floats, and booleans, OCaml provides built-in support for creating tuples, strings, lists, arrays, and optionals.

```
let _x0 = (1, 2, 3) in              (* Tuples *)
let _x1 = "hello, world" in         (* Strings *)
let _x2 = [1 ; 2 ; 3 ; 4 ] in       (* Lists *)
```

```
let _x3 = [| 1 ; 2 ; 3 ; 4 |] in     (* Arrays *)
let _x4 = Some 4 in                  (* Optionals *)
let _x5 = None in
() ;;
```

2.4.3 User defined types

Users can define their own types. OCaml provides three main constructs for doing so: type aliasing, type constructors, and records.

You can give an alternative name to an existing type using a *type alias*.

```
type my_int = int
type my_int_list = int list ;;
```

Multiple types can be grouped together using *variants*. A variant type defines a set of constructors for initializing an object of that type. As an example, I can define an integer cons-list as follows:

```
type int_cons_list =
  | Cons of int * int_cons_list
  | Nil
let _x = Cons 1 (Cons 2 (Cons 3 Nil)) in () ;;
```

You can also group types using *records*.

```
type my_user = {
  name: string ;
  age: int ;
  height: float ;
}
let _x = { user= "Foo" ; age= 42 ; height= 3.14 } in () ;;
```

2.5 Polymorphism

Our cons-list definition in the previous section was specialized to integers. However, can make generalize the definition using *polymorphic types*.

```
type cons_list 'a =
  | Cons of 'a * cons_list
  | Nil
let _int_list = Cons 1 (Cons 2 (Cons 3 Nil)) in
let _string_list = Cons "a" (Cons "b" (Cons "c" Nil)) in () ;;
```

This form of polymorphism is known in the literature as *parametric polymorphism*. Rest assured this is not the only form of polymorphism available in OCaml — I omit further discussion for brevity.

2.6 Pattern matching

Like most other functional programming languages, OCaml also supports *pattern matching*. This allows programmers to destruct objects of a given type into its constituent parts based on its shape.

```
let rec my_map f xs =
  match xs with
  | Cons (x, rest) -> Cons (f x, map f rest)
  | Nil -> Nil
let rec map f xs =
  match xs with
  | [x ; rest] -> (f x) :: (map f rest)
  | [] -> []
;;
```

2.7 Imperative constructs

Despite being a functional programming language, OCaml offers several facilities for writing imperative programs.

You can create a mutable value using a *ref*.

```
let x = ref 0 in          (* Creation *)
x := 2 ;                  (* Mutation *)
!x ;;                     (* Access *)
```

Note how the imperative update (`x := 2`) was sequenced with an another expression using a semi-colon (`;`).

The fields of a record can also be defined to be mutable.

```
type my_mutable_user = {
  name: string ;
  mutable age: int ;
  mutable height: float ;
}
let x = { name= "Bar"; age= 42; height= 3.14 } in
x.age <- 44 ;
x.height <- 6.28 ;;
```

Finally, OCaml also has built-in (albeit rather limited) support for `while`-loops and `for`-loops.

```
let x = [| 0 ; 0 ; 0 ; 0 ; 0 |] in
let num_elements = Array.length x in

let i = ref 0 in
while !i < num_elements do
  x.(!i) <- !i              (* Array elements are mutable too! *)
done ;

```

```
for j = 0 to num_elements - 1 do
  x.(j) <- j
done ;;
```

2.8 Exceptions

OCaml has built-in support for exceptions. Certain design decisions in the compiler (namely the ABI) are made to make exceptions fast and cheap (Section 3.2.3).

```
(* Defining a custom exception *)
exception MyBad of string

(* Raising an exception *)
let will_raise_my_bad () =
  raise (MyBad "oops")

(* Handling exceptions *)
let () =
  try will_raise_my_bad () with
  MyBad s -> print_string s
```

2.9 Modules

One of the distinguishing features of OCaml is the module system. Modules are an abstraction for packaging code and defining interfaces.

By default, all code within a file lives inside a module of the same name as the file. For example, suppose we put the following code in `foo.ml`:

```
let my_foo a b = a + b
```

Then, you can use `my_foo` in a separate file `bar.ml` as follows:

```
let my_bar = Foo.my_foo 5 10
```

Modules can be *opened*, which will make all the names in that module available in the current scope.

```
open Foo

(* Do not need to qualify `my_foo` *)
let my_bar = my_foo 5 10
```

Modules can be nested. You can, for example, define another module inside a file-based module as follows:

```
module NestedFoo = struct
  let my_nested_foo a b = a + b
end

let my_foo = NestedFoo.my_nested_foo 5 10
```

Programmers can control the interface exposed by a module using a *signature*. The interface contains information on the symbols exposed by the module and their types. For file-based modules, the signature is specified using an interface file, having the `mli` file extension. For example, we can define a signature for `foo.ml` in a `foo.mli` file as follows:

```
val my_foo : int -> int -> int
```

Signatures can also be defined inside a file.

```
sig NestedFooIface = sig
  val my_nested_foo : int -> int -> int
end

module NestedFoo : NestedFooIface = struct
  let my_nested_foo a b = a + b
```

```
end
```

In addition to specifying interfaces, signatures can also be used to define *abstract types*. These are used to hide the underlying implementation details of a module, which can be very useful in controlling how a module can be used.

```
module type AbstractStackIface = sig
  type 'a t

  val init : unit -> 'a t
  val push : 'a t -> 'a -> 'a t
  val pop  : 'a t -> ('a option * 'a t)
end

module AbstractStack : AbstractStackIface = struct
  type 'a t = 'a list

  let init () = []

  let push xs x = x :: xs

  let pop xs =
    match xs with
    | [] -> (None, [])
    | x :: rest -> (Some x, rest)
end
```

In the above snippet, we are defining an `AbstractStack` module that hides the fact that the stack is implemented using a list by abstracting away the `'a t` type in the `AbstractStackIface` signature.

2.10 Further Reading

This chapter provided a high-level overview of some of the prominent features of OCaml. For readers interested in learning more, I recommend taking a look at the OCaml manual ([3]) which contains information on all you would ever need to know about the language presented in an accessible way.

CHAPTER 3

THE OCAML COMPILER

The OCaml compiler has a long history, tracing back to its origins in the ZINC compiler [4]. In this thesis, I will be exclusively discussing version 4.14.0 of the compiler. Note that this version does not include the (recently upstreamed) multi-core version of OCaml [5].

The compiler is open-source and is readily available at https://github.com/ocaml/ocaml, along with the OCaml runtime system. The codebase consists of approximately 388k lines of code.

The compiler has two variants — bytecode and native. The bytecode compiler produces compact, portable code that can be interpreted by the runtime system. It is fast but not as performant as the native compiler, which generates high-performance machine code that can be directly run on the target system. It supports several platforms, including x86 (32 and 64 bit), ARM (32 and 64 bit), Power (32 and 64 bit), RISC-V (64 bit), and IBM Z (s390x).

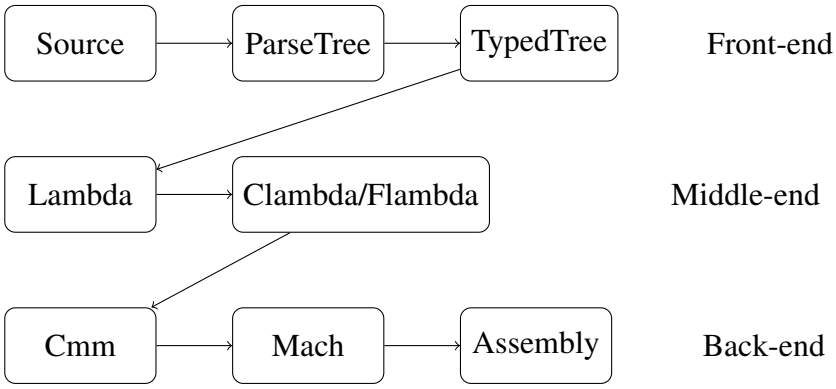


Figure 3.1: OCaml Compilation Pipeline

```
let rec map f xs =
  match xs with
  | [] -> []
  | x :: rest -> (f x) :: (map f rest)
;;
```

(a) OCaml

```
(setglobal Map!
  (letrec
    (map/267
       (function f/268 xs/269
         (if xs/269
           (makeblock 0 (apply f/268 (field 0 xs/269))
             (apply map/267 f/268 (field 1 xs/269)))
           0)))
    (makeblock 0 map/267)))
```

(b) Lambda

Figure 3.2: OCaml code for the `map` function over lists, represented in Lambda form

3.1 Compilation Pipeline

Fig. 3.1 summarizes the OCaml compilation pipeline. Compilation starts with parsing, which converts OCaml source code into ParseTree, which is an untyped abstract syntax tree (AST) contained detailed information about the structure of the program. Lexing is done using the ocamllex lexer generator [3] and parsing is done using the menhir parser generator [6].

The ParseTree AST is then fed into the typechecker which runs type inference and checks the program for typing errors. If it succeeds, it outputs a typed AST called TypedTree. For a detailed treatment of OCaml's type checker, refer to Rémy [7], Wright and Felleisen [8], and Kiselyov [9].

After type checking, TypedTree is lowered into a smaller, untyped calculus called Lambda (Fig. 3.2). The lowering gets rid of type annotations, replacing some of them with *value kinds*, which encode information about the memory representation of a value (generic, int, boxed int, or float). Further, high-level constructs like modules and classes

```
let x = Random.int 10 in
let f y = x + y in
f 5
```

(a)

```
let x = Random.int 10 in
let f y env =
  (field env 0) + y
in
f 5 (env with { 0 => x })
```

(b)

Figure 3.3: *Implicit closures*. In this example, the binding `x` is implicitly captured by the closure `f`. To make this capture explicit, the compiler will add an extra `env` parameter. The closure can then using this parameter to access the captured value of `x`.

```
(* Direct call *)
let f x = x + 1 ;;
f 13 ;;

(* Generic call *)
(let y = 2 in
 (fun x -> x + y)) 13 ;;
```

Figure 3.4: *Implicit function calls*. In this example, the first function call is a *direct call*, using an explicit label to indicate the function being invoked. On the other hand, the second function call is a *generic call*, using a generic expression that returns a lambda, in-place of a function label.

get compiled down to records and function pointers. Pattern matching is also compiled away to optimized Lambda code using a technique presented by Le Fessant and Maranget [10]. The compiler runs two optimization passes on Lambda: tail-modulo cons (TMC) [11] and Simplif. TMC aids in optimizing away tail calls while Simplif tries to simplify expressions by getting rid of unnecessary let-bindings through inlining.

Lambda form, however, is not amenable to code generation. First, closures are still implicit (Fig. 3.3a). Machine code has no notion of capturing variables, let alone closures. The compiler will have to generate code to invoke the closure in the correct environment (Fig. 3.3b). Second, function calls are not explicit (Fig. 3.4). By classifying function calls, the compiler can make better optimization decisions. For example, it can decide to inline a generic call if it is small enough.

The middle-end serves as an intermediary prior to handing control over to the back-

end for machine code generation. OCaml has two middle ends to choose from. The default is Clambda, which performs closure conversion and makes function calls explicit. Closure conversion makes variable capture explicit by passing an extra environment argument which closures can use to refer to captured variables. The alternative middle-end is Flambda, which in addition will perform several optimizations like more aggressive inlining, function specialization, unboxing, and code lifting. A discussion of these optimizations is beyond the scope of this thesis. The interested reader should refer to the manual [3] for more details.

Now, the backend is invoked. OCaml's backend consists of two main IRs: Cmm (not to be confused with C−− [12]) and Mach. Cmm is a machine-independent IR that bears a striking resemblance to Lambda/Clambda, although it differs in that primitives and function applications are broken down into fine-grained operations. When converting to Cmm, value kind information is propagated, which serves two purposes: (1) guiding the register allocator (for example, integer values go into integer registers), and (2) aiding the GC in local variable tracking (for example, unboxed values can be ignored). Instruction selection is then performed to generate Mach, a machine-specific IR. The compiler applies standard optimization passes like common sub-expression elimination (CSE), and dead code elimination (DCE), along with a CombAlloc pass, which coalesces allocation requests. Lastly, register allocation and instruction scheduling are executed, followed by linking the resulting binary with the runtime system.

3.2 The Runtime System

OCaml's runtime system is written in C and consists of about 29k lines of code. It implements primitives, the garbage collector, and a bytecode interpreter. It also specifies the memory representation of OCaml values. Since OCaml programs follow a different application binary interface (ABI) compared to C, using runtime routines requires the program to switch to and from C.

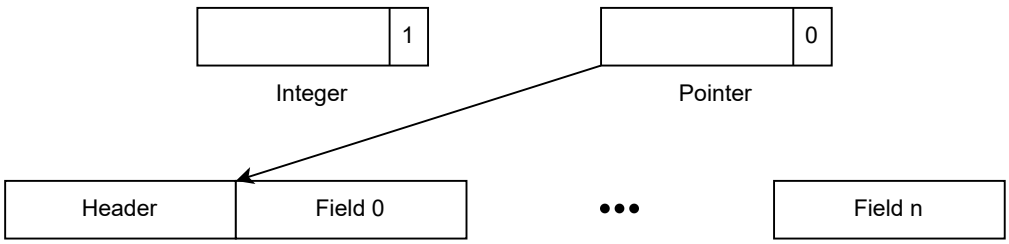


Figure 3.5: Representation of integers and pointers

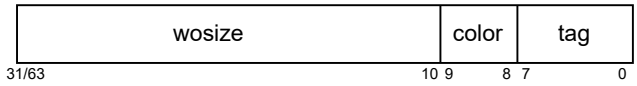


Figure 3.6: Representation of the 32/64-bit header

3.2.1 Memory representation of Values

Values in OCaml are denoted using tagged pointers, where the least significant bit is utilized to differentiate between pointers and integers. When the least significant bit is set to one, the value is an integer; otherwise, it is a pointer. Pointers point to a block which includes a header and its corresponding fields (Fig. 3.5). The header (Fig. 3.6) is made up of three parts:

- *wosize*: The number of words in a block (excluding the header)
- *color*: Used by the garbage collector during the mark-sweep phase
- *tag*: Used to determine the type of the block

For example, floats are represented as boxed values containing a single field (Fig. 3.7).

Figure 3.7: Memory representation of floats

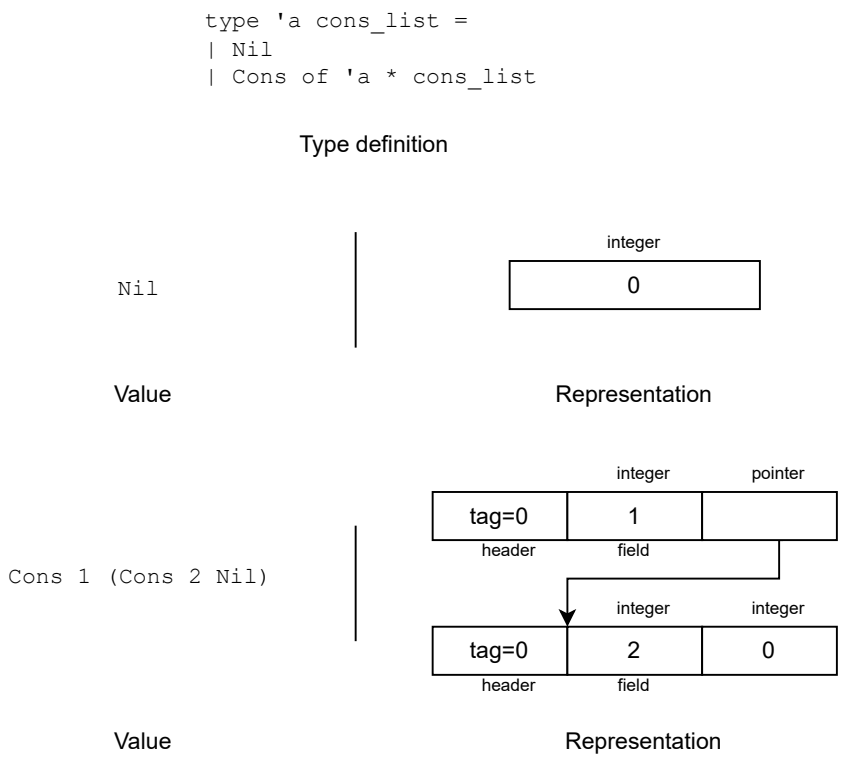


Figure 3.8: Memory representation of variants

Variants, on the other hand, have a different encoding (Fig. 3.8). Variants with no parameters are encoded as unboxed integers, starting from zero. Those that have parameters, however, are encoded as boxed values with tags starting from zero. Lists are a special case of variants: the empty list is represented as the integer zero, while non-empty lists are represented as a boxed value with two fields. The first field contains the value of the list's head, and the second field points to the rest of the list.

The runtime utilizes various techniques to efficiently encode specific types of values. Two examples of such values are float arrays and closures.

Generally, an array is encoded as a boxed value with the fields representing the elements of the array. Note that floats are boxed which means that accessing the elements of a float array require two levels of indirection: one to access the address of the element, and another to access the float value of the element. To make using float arrays more efficient, the runtime gets rid of the second indirection by packing the float values in the fields, i.e., by

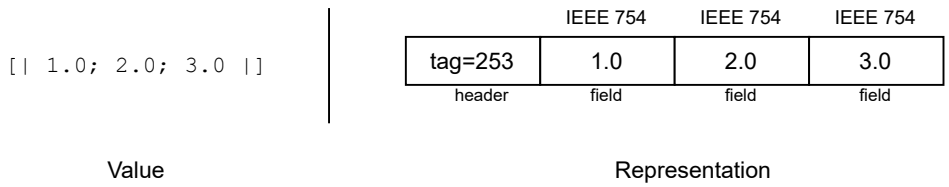


Figure 3.9: Memory representation of float arrays. Note how the elements are packed instead of pointing to separate float blocks.

storing them as unboxed floats Fig. 3.9.

Closures in OCaml can be mutually recursive. To save space, OCaml will encode a set of mutually recursive closures in a single block with a common shared environment (Fig. 3.10) [13]. The general layout consists of a header followed by an encoding of the closures (each separated by an *infix header*) followed by an encoding of the environment. The individual closures themselves can have two separate encodings depending on their arity. If the function arity is one, the encoding consists of a header, followed by a pointer to the code for the closure, followed by an integer field encoding the arity and the offset to the start of the environment (Fig. 3.11a). Otherwise, the encoding also contains a pointer to a curried version of the closure (Fig. 3.11b).

3.2.2 Garbage Collection

OCaml uses a generational GC (which is a type of tracing GC) that manages two heaps: minor and major. The design is based on the principle that most objects are small and short-lived. Further, the GC is synchronous and can get invoked only during allocation requests or polling sites (both of which are statically inserted by the compiler).

The minor heap is small, about 256K words by default. New objects are allocated in the minor heap using a bump allocator. Allocations in the minor heap are cheap (for example, the x86 runtime uses about four assembly instructions on average for a minor heap allocation). When the minor heap is full, a copying collector will promote live objects to the major heap. It works by first promoting all the roots followed by transitively promoting

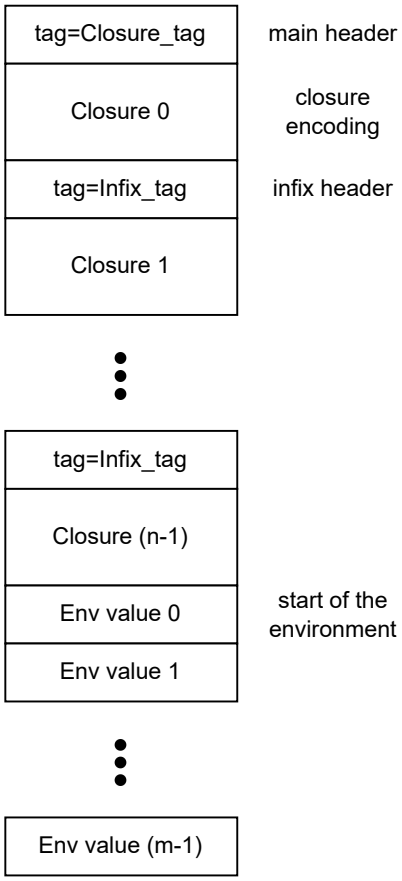
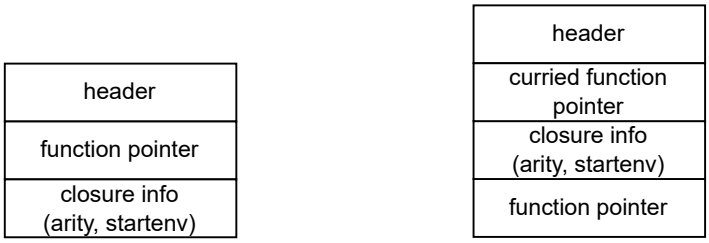


Figure 3.10: Memory representation of a set of n mutually recursive closures collectively capturing m variables in the environment



(a) Arity = 1 (b) Arity > 1

Figure 3.11: Memory representation of an individual closure

their children. What remains in the minor heap is reused for further allocation. Since objects are small and short-lived, the copying collector does not take much time to run and thereby minimizes pause times.

The major heap is larger, containing promoted objects as well as objects that could not fit in the minor heap. It uses a sophisticated allocation mechanism, using a software managed paging system backed by libc's `malloc` and `free` routines. When the major heap is full, collection is carried out using an incremental mark-and-sweep approach. Objects can be marked with four possible colors, encoded as part of the object header:

- *black*: Indicates that the object is live and must not be freed
- *white*: Indicates that the object is dead (i.e., there are no references to it) and can be freed
- *grey*: Indicates objects that are alive but their children have not been scanned completely
- *blue*: Indicates that the block is free memory not encoding an object and can be used for future allocations

The marking phase maintains a stack of objects to be traversed. It starts by first marking the roots grey and pushing them onto the stack. Then, marking is continued by popping the stack, marking the children of the popped object, and then marking the object itself black. This process continues until the mark stack becomes empty.

The sweeping phase traverses the heap and turns white objects blue (indicating that they are available for allocation) and black objects white (to reset before a future scan).

Both the phases are carried out in an incremental fashion, interleaved with the user program. As a result, the collector must also be aware of new allocations and mutation. If a new allocation happens during the marking phase, it is colored black. If it happens during the sweeping phase, the is colored black or white depending on whether sweeping

has reached the new allocation. Mutation is handled by invoking a *write barrier* that loads the mutated object and marks it grey if it is white, adding it to the mark stack if needed.

3.2.3 Calling into the runtime: x86 edition

First, a discussion of OCaml's ABI is in order. There are three calling conventions to be aware of, summarized in Fig. 3.12:

- *C calling convention*: This is the standard *cdecl* calling convention, used by many x86 C compilers. Certain registers are callee-saved.
- *OCaml calling convention*: In this convention, all registers are caller-saved. This is makes exception handling extremely cheap. However, it is incompatible with the C calling convention.
- *Allocation calling convention*: This convention is used by the allocation routines. It does not require the caller to save any registers. In the fast case, only the bump allocator is invoked, implemented with a single pointer bump followed by a check to trigger the garbage collector if needed.

In addition to this, OCaml also pins the `r14` and `r15` registers to point to the exception and bump allocator pointers respectively.

OCaml programs call into the C runtime by performing a runtime switch. Due to the incompatibility in calling conventions, the runtime switch will need to save and restore a slew of registers, making the whole process quite expensive Fig. 3.13 shows the assembly that is run to invoke the garbage collector, which is implemented in the C runtime.

While the details mentioned here are specific to x86, the knowledge is easily portable to make sense of how it is done for the remaining backends.

C

Arguments:
integer: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
float: `xmm0` – `xmm7`

Results:
integer: `rax`, `rdx` float: `xmm0`, `xmm1`

Notes:
Same as the *cdecl* convention used in C compilers. Registers `rbx`, `rbp`, `r12`-`r15` are callee-saved.

OCaml

Arguments:
integer: `rax` - `r13`
float: `xmm0` – `xmm9`

Results:
integer: `rax`
float: `xmm0`

Notes:
All registers are caller-saved, with registers `r14` and `r15` pinned to global values.

Allocation

Arguments:
Same as OCaml

Results:
Same as OCaml

Notes:
Used for allocation in the minor heap, and in the worst-case can invoke the GC.

Figure 3.12: OCaml ABI summary

```
        pushq   %r11; CFI_ADJUST (8);
        pushq   %r10; CFI_ADJUST (8);
        pushq   %r13; CFI_ADJUST (8);
        pushq   %r12; CFI_ADJUST (8);
        pushq   %r9; CFI_ADJUST (8);
        pushq   %r8; CFI_ADJUST (8);
        pushq   %rcx; CFI_ADJUST (8);
        pushq   %rdx; CFI_ADJUST (8);
        pushq   %rsi; CFI_ADJUST (8);
        pushq   %rdi; CFI_ADJUST (8);
        pushq   %rbx; CFI_ADJUST (8);
        pushq   %rax; CFI_ADJUST (8);
        movq    %rsp, Caml_state(gc_regs)
    /* Save young_ptr */
        movq    %r15, Caml_state(young_ptr
    /* Save floating-point registers */
        subq    $(16*8), %rsp; CFI_ADJUST
        movsd   %xmm0, 0*8(%rsp)
        movsd   %xmm1, 1*8(%rsp)
        movsd   %xmm2, 2*8(%rsp)
        movsd   %xmm3, 3*8(%rsp)
        movsd   %xmm4, 4*8(%rsp)
        movsd   %xmm5, 5*8(%rsp)
        movsd   %xmm6, 6*8(%rsp)
        movsd   %xmm7, 7*8(%rsp)
        movsd   %xmm8, 8*8(%rsp)
        movsd   %xmm9, 9*8(%rsp)
        movsd   %xmm10, 10*8(%rsp)
        movsd   %xmm11, 11*8(%rsp)
        movsd   %xmm12, 12*8(%rsp)
        movsd   %xmm13, 13*8(%rsp)
        movsd   %xmm14, 14*8(%rsp)
        movsd   %xmm15, 15*8(%rsp)
    /* Call the garbage collector */
        PREPARE_FOR_C_CALL
        call    GCALL(caml_garbage_collect
```

(a) Setup and invocation

```
        CLEANUP_AFTER_C_CALL
    /* Restore young_ptr */
        /* movq    Caml_state(young_ptr), %r15 */
    /* Restore all regs used by the code generator */
        movsd   0*8(%rsp), %xmm0
        movsd   1*8(%rsp), %xmm1
        movsd   2*8(%rsp), %xmm2
        movsd   3*8(%rsp), %xmm3
        movsd   4*8(%rsp), %xmm4
        movsd   5*8(%rsp), %xmm5
        movsd   6*8(%rsp), %xmm6
        movsd   7*8(%rsp), %xmm7
        movsd   8*8(%rsp), %xmm8
        movsd   9*8(%rsp), %xmm9
        movsd   10*8(%rsp), %xmm10
        movsd   11*8(%rsp), %xmm11
        movsd   12*8(%rsp), %xmm12
        movsd   13*8(%rsp), %xmm13
        movsd   14*8(%rsp), %xmm14
        movsd   15*8(%rsp), %xmm15
        addq    $(16*8), %rsp; CFI_ADJUST(-16*8)
        popq    %rax; CFI_ADJUST(-8)
        popq    %rbx; CFI_ADJUST(-8)
        popq    %rdi; CFI_ADJUST(-8)
        popq    %rsi; CFI_ADJUST(-8)
        popq    %rdx; CFI_ADJUST(-8)
        popq    %rcx; CFI_ADJUST(-8)
        popq    %r8; CFI_ADJUST(-8)
        popq    %r9; CFI_ADJUST(-8)
        popq    %r12; CFI_ADJUST(-8)
        popq    %r13; CFI_ADJUST(-8)
        popq    %r10; CFI_ADJUST(-8)
        popq    %r11; CFI_ADJUST(-8)
```

(b) Cleanup

Figure 3.13: Invoking the GC by switching into the C runtime. Some portions have been omitted for brevity. Note the number of registers that have to be save/restored.

CHAPTER 4

PERCEUS: GARBAGE FREE REFERENCE COUNTING WITH REUSE

Managing memory is a resource allocation problem, as a computer has a limited amount of memory that needs to be shared among programs with varying memory needs. Memory management occurs at different levels of the computing stack, including at the hardware, operating system, or application levels. In this chapter, we will focus on the memory management abstractions provided by programming languages.

Programming languages offer different facilities for managing memory within the address space defined by the operating system. These can be broadly classified as manual or automated. In manual memory management, the programmer has complete control over a pool of memory and must explicitly specify when the system can reclaim memory. In contrast, automated techniques shift this responsibility onto runtimes, which abstract away memory management by automatically inferring when and where objects should be allocated or deallocated.

4.1 Manual memory management

This memory management technique is commonly used in systems programming languages that prioritize granting the programmer greater control to build high-performance systems. Here, the programming language provides interfaces for allocating and deallocating memory, which the program must explicitly invoke to meet memory requests. These interfaces offer precise control over memory usage by abstracting over the operating system's system calls (such as `sbrk` and `mmap`), which manage memory in larger, page-sized chunks. For instance, the `malloc` function in libc employs a free list to handle application memory requests using memory obtained from the operating system.

Manual memory management has several advantages. It enables programmers to write

```
int main() {
  int abort = 0;
  void *x = malloc(0x1000);
  int err = process(x);
  if (err) {
    abort = 1;

    // x is freed and abort
   is set to 1
    free(x);
  }
  ...
  if (abort) {
    // x is used after being
     freed!
    printf("Error: %p", x);
  }
}
```

(a) Use after free

```
int main() {
  void *x = malloc(0x1000);
  int err = process(x);
  if (err) {
    // x is first freed here
    free(x);
    goto fail
  }
  ...

fail:
  // x is freed again!
  free(x);
}
```

(b) Double free

Figure 4.1: Manual memory management bugs. While the examples seen innocuous, they are exploitable vulnerabilities.

high-performance code by eschewing the runtime overhead of using automated techniques like tracing or reference counting. This, in turn, allows the programmer to give better guarantees on properties like latency and resource usage, which are especially important in time-critical, resource-constrained systems. By ceding control to the programmer, users have the flexibility of deploying customized memory management schemes that are optimized for their particular workloads.

However, the pastures are not all green in the land of manual memory management. It is more likely that users misuse the control afforded by the technique, which result in wasteful memory use and unexpected performance hits. The two most common bugs in this sphere are use after free (Fig. 4.1a) and double free (Fig. 4.1b). Modern systems programming languages address this issue by enforcing a paradigm that makes such bugs harder to manifest in common usage. An example of such a paradigm is Resource Allocation is Initialization (RAII), popularized by C++. Note that these paradigms do not preclude fine-grained control — the programmer can opt out of using the paradigm.

```
int main() {
  void *x = malloc(0x1000);
  init(x);
  process(x);
  print(x);

  // x can be freed here

  void *y = malloc(0x1000);
  init(y);
  process(y);
  print(y);

  // However, it is freed much later
  free(x);
  free(y);
}
```

Figure 4.2: Holding onto memory for longer than needed, a notable flaw of manual memory management

Another issue, however, is that resources can be held for longer than needed Fig. 4.2. In such cases, it would help to have a compiler pass analyze the lifetimes and emit precise allocation and deallocation requests, treading into the sphere of automated memory management.

4.2 Automated memory management

Automated memory management encompasses techniques that abstract away the concerns of memory management from the programmer, shifting the burden onto language runtimes. Broadly, they can be classified into two categories: tracing and reference counting.

4.2.1 Tracing

In tracing based systems, a garbage collector (GC) traces the heap to determine which objects are alive and therefore should not be deallocated It does this by first tracing the roots, which are a list of objects known to be alive at the start of a collection, continuing until it has traced all objects transitively referenced by the roots. There are many ways to

implement tracing collectors.

The most naive way, as exemplified by the explanation prior, is referred to as mark-and-sweep. Here, collection is split into two phases: marking and sweeping. During the mark phase, the collector will trace the heap, marking live objects as alive. During the sweep phase, the collector will deallocate all dead objects. While simple to implement, the main disadvantage is that it pauses the program execution during collection (colloquially referred to as *stopping the world*), which can result in unexpected freezes in program execution, thereby making it unsuitable for time-critical systems.

To address the pauses, mark-and-sweep is augmented with a tri-color abstraction in which objects are marked dead (typically colored white), alive with no outgoing references to traverse (typically colored black), and alive with outgoing references that have not been traversed (typically colored grey). By using three colors, the marking and sweeping phases can now be performed incrementally, thereby mitigating unpredictable pause times. This should remind you of our discussion on OCaml's GC, which in fact uses the tri-color abstraction.

Apart from pauses, a tracing GC may also have to deal with fragmentation. There are two choices here:

- Leave all live objects as is. Such a collector is referred to as a non-moving (or non-compacting) collector.
- Move live objects to a new area of memory. Such a collector is referred to as a moving (or compacting) collector.

The main benefit of a non-moving collector is that the overhead of a collection is lowered. However, moving collectors can improve overall program performance by reducing allocation/deallocation time and improving spatio-temporal locality of objects.

It has been empirically observed in programs that the most recently allocated objects are the most likely to be unreachable quickly. This motivates the idea of splitting the heap into

generations ranging from youngest to oldest, promoting objects that survive a collection to the older generation. Such collectors are referred to as generational or ephemeral collectors. In practice, they can result in faster, incremental collection cycles.

Tracing collectors can also execute in multi-threaded environments. In such cases, careful attention needs to be paid to how the different phases can execute concurrently with the user program.

4.2.2 Reference counting

Reference counting is a simple memory management technique in which an object is freed as soon as its reference count drops to zero. It can either be implemented manually (where the programmer explicitly manages reference counts) or automatically (where a a compiler or interpreter manages the reference counts on behalf of the programmer).

One of the main advantages of reference counting over tracing is that it is relatively simple to implement. Further, it can lead to better memory utilization because objects are freed as soon as they are no longer referenced. The cost of tracing is linear in live data, while the cost of reference counting is linear in the number of reference counting operations. This means that the performance of reference counting does not deteriorate as the number of live objects increases. Reference counting does not cause stop-the-world pauses due to its incremental nature. The reference counts themselves can serve as useful data for performing certain runtime optimizations. For instance, if it is known that an object is unique (i.e., its reference count is one), then the compiler can perform in-place updates on the object. This is an especially important optimization in functional programming languages where objects are immutable by default. Another optimization that is enabled by reference counting is reuse. Here, the system tries to reuse the memory from a prior deallocation instead of returning back to the free pool, making allocation in this case free. We talk more about reuse in Section 4.3.

However, there are certain problems that make reference counting expensive in practice:

- *Frequency of updates*: Maintaining reference counts is not free. In a naive implementation, it is possible that the program will have to perform a large number of these updates. Tracing collectors, in practice, have a lower amortized cost as they are typically run infrequently.

- *Cycles*: A more insidious issue has to do with reference cycles — two or more objects refer to each other in a cyclic fashion, which results in neither of their reference counts to drop to zero. This can particularly arise in cyclic data structures. This issue is typically addressed by using *weak reference counts* or resorting to a tracing collector.

- *Concurrency*: In the presence of shared data structures, the reference count operations need to be atomic, which incurs additional costs.

- *Precision*: Reference counting systems may even be imprecise, holding on to objects for longer than necessary. This can happen when reference counting operations are performed at the end of lexical scopes (usually done to make exception handling and stack unwinding simpler), as shown in Fig. 4.2.

4.3 Perceus

In 2021, Reinking *et al.* [1] presented their work on Perceus, a reference counting scheme with excellent performance characteristics (Section 4.3) compared to mature memory collectors like that of Haskell, OCaml, and Swift. Common reference counting implementations, as exemplified in the previous section, might retain memory longer than needed. Perceus does away with this issue by emitting *precise* reference counting operations, freeing an object *as soon as* no more references remain. This, in turn, makes cycle-free programs *garbage free*, where only live references are retained at any point in the program.

As an example, consider:

```
1 let foo () =
```

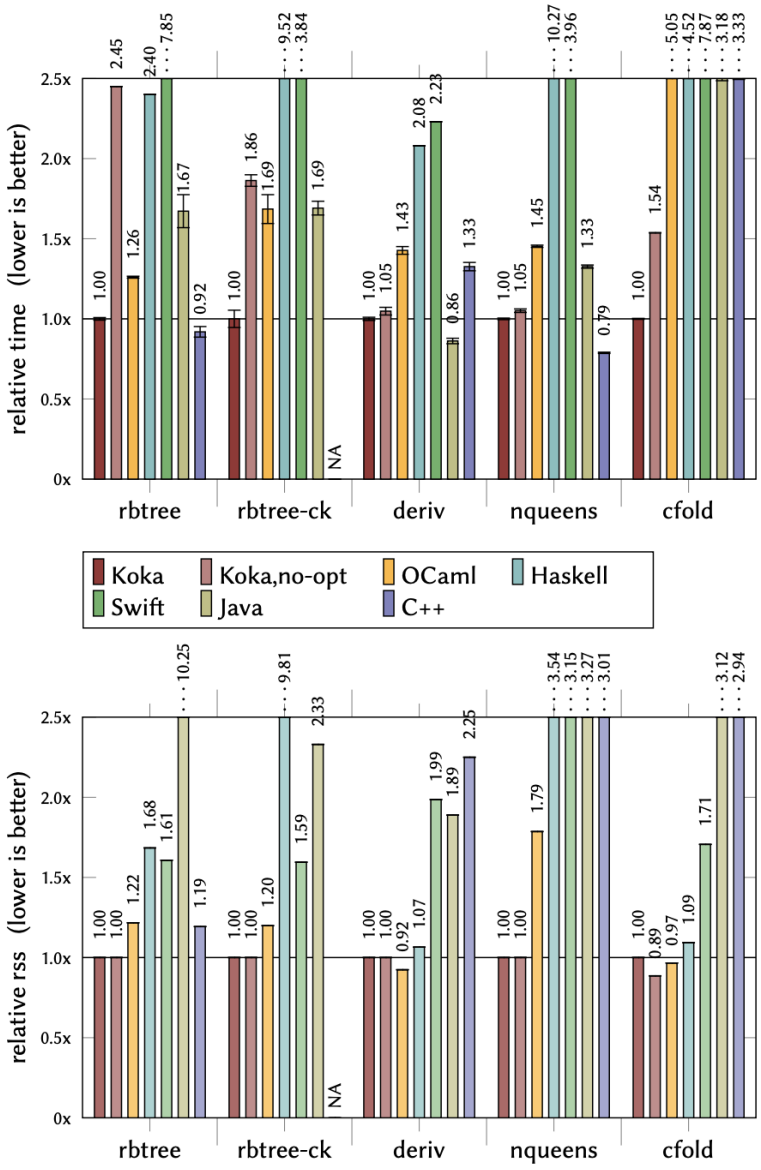


Figure 4.3: Relative execution time and peak working set with of Perceus as implemented in Koka [1].

```
2   let xs = range 0 100 in (* Create a list *)
3   let ys = map f xs in    (* Map a function `f` over it *)
4   print ys
```

A traditional reference counting system that tracks liveness based on lexical scope would emit the following:

```
1 let foo () =
2   let xs = range 0 100 in (* Create a list *)
3   let ys = map f xs in    (* Map a function `f` over it *)
4   print ys ;
5   drop xs ;
6   drop ys
```

Note how the memory held by `xs` is still live during the execution of `print` even though it is not referenced at or after this point.

In Perceus, no `drop` instructions will be emitted in `foo`. Instead, ownership of the references to `xs` and `ys` is passed down into `map` and `print` respectively, which are now responsible for emitting the drops. At first glance, this approach may seem expensive. However, Perceus enables further optimization opportunities, like *drop specialization* and *reuse specialization*. Additionally, Perceus enables a novel programming paradigm called functional but in-place (FBIP), which lets programmers write in-place mutating algorithms in a functional way.

Let us examine these optimizations by considering the `map` function (Fig. 4.4a). After running Perceus, `map` will emit `drop` instructions for `xs` and `f` since it takes ownership of its arguments (Fig. 4.4b). Drop specialization (Fig. 4.4c) inlines the definition of `drop` (Fig. 4.6), which when coupled with fusion results in almost no reference counting operations in the fast path (Fig. 4.4d). *Reuse analysis* builds upon drop specialization by tracking which freed memory regions can be reused by future allocations. It does this by first producing *reuse tokens* using the `drop-reuse` primitive (Fig. 4.7), each of which is linked to

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list

let rec map f xs =
  match xs with
  | Nil -> Nil
  | Cons (x, rest) -> Cons (f x, map f rest)
```

(a) The `map` function

```
let rec map f xs =
  match xs with
  | Nil -> ( drop xs ; drop f ; Nil )
  | Cons (x, rest) -> (
    dup x ;
    dup rest ;
    drop xs ;
    Cons ((dup f) x, map f rest)
  )
```

(b) After inserting reference counting instructions

```
let rec map f xs =
  match xs with
  | Nil -> ( drop xs ; drop f ; Nil )
  | Cons (x, rest) -> (
    dup x ;
    dup rest ;
    if is_unique xs then (
      drop x ;
      drop rest ;
      free xs
    ) else (
      decref xs
    ) ;
    Cons ((dup f) x, map f rest)
  )
```

(c) After drop specialization

```
let rec map f xs =
  match xs with
  | Nil -> ( drop xs ; drop f ; Nil )
  | Cons (x, rest) -> (
    if is_unique xs then (
      free xs
    ) else (
      dup x ;
      dup rest ;
      decref xs
    ) ;
    Cons ((dup f) x, map f rest)
  )
```

(d) After fusion

```
let rec map f xs =
  match xs with
  | Nil -> ( drop xs ; drop f ; Nil )
  | Cons (x, rest) -> (
    dup x ;
    dup rest ;
    let ru = drop_reuse xs in
    Cons@ru ((dup f) x, map f rest)
  )
```

(e) Reuse token insertion

```
let rec map f xs =
  match xs with
  | Nil -> ( drop xs ; drop f ; Nil )
  | Cons (x, rest) -> (
    dup x ;
    dup rest ;
    let ru =
      if is_unique xs then (
        drop x ;
        drop rest ;
        &xs
      ) else (
        decref xs ;
        NULL
      )
    in
      Cons@ru ((dup f) x, map f rest)
  )
```

(f) Drop specialization for the `drop-reuse` primitive

```
let rec map f xs =
  match xs with
  | Nil -> ( drop xs ; drop f ; Nil )
  | Cons (x, rest) -> (
    let ru =
      if is_unique xs then (
        &xs
      ) else (
        dup x ;
        dup rest ;
        decref xs ;
        NULL
      )
    in
      Cons@ru ((dup f) x, map f rest)
  )
```

(g) After fusion

Figure 4.4: Perceus on `map`, including drop specialization and reuse analysis

```
type t =
  | Foo of int * int * t
  | Bar of int
  | Baz

let foo n =
  match n with
  | Foo (a, b, c) -> (
      dup c ;
      let ru = drop_reuse n in
      Foo@ru (a + 1, b, c)
  )
  | _ -> n
```

(a) Original program, after reuse token insertion

```
if ru <> NULL then (
  (* In-place, fast path *)
  ru[0] := a + 1 ;
  ru[1] := b ;
  ru[2] := c ;
  ru
) else (
  (* Fresh allocation *)
  Foo (a + 1, b, c)
)
```

(b) `Foo@ru` expanded, before reuse specialization

```
if ru <> NULL then (
  (* In-place, fast path *)
  ru[0] := a + 1;
  ru
) else (
  (* Fresh allocation *)
  Foo (a + 1, b, c)
)
```

(c) `Foo@ru` expanded, after reuse specialization

Figure 4.5: Reuse specialization in action. Note how after specialization, fields 1 and 2 of `Foo` are reused in the fast path, resulting in only one assignment (as opposed to three). `ru[i]` refers to the ith field of a constructor represented by `ru`.

```
let drop x =
  if is_unique x then (
    drop children of x ;
    free x
  ) else (
    decref x
  )
```

Figure 4.6: Pseudocode for the `drop` primitive

```
let drop_reuse x =
  if is_unique x then (
    drop children of x ;
    &x
  ) else (
    decref x ;
    NULL
  )
```

Figure 4.7: Pseudocode for the `drop-reuse` primitive

a compatible constructor that can reuse the allocation represented by the token if possible. Now, in the fast path, the memory of `xs` can be reused (Fig. 4.4f and Fig. 4.4g), effectively updating the list in-place. Reuse specialization leverages reuse analysis to further try reusing the unchanged fields of a constructor. `map` does not benefit from this optimization as all the fields are assigned. Instead, consider Fig. 4.5 in which all the fields of the `Foo` constructor except the first one remain unchanged. Reuse specialization can identify this and eliminate unnecessary assignments.

Table 4.1 summarizes all of the primitives used by Perceus. I refer readers to the Perceus technical report ([14]) for more detailed information about the algorithm.

Table 4.1: Reference counting primitives used by Perceus and its optimizations

Primitive	Description
`dup`	Increments the reference count of an object by one
`drop`	Decrements the reference count of an object by one. If it drops to zero, recursively drops all children of the object and frees its memory
`drop-reuse`	Same as `drop`, but returns a reuse token instead of freeing its memory
`decr`	Decrements the reference count of an object by one
`is-unique`	Returns true if the reference count of an object is one
`alloc`	Allocates a block of memory
`free`	Frees a block of memory
`Foo@ru`	Allocates a constructor `Foo`, reusing memory tagged with the reused token `ru` when possible

CHAPTER 5

IMPLEMENTATION

Perceus was implemented on version 4.14.0 of the OCaml compiler, in roughly 2.8k lines of code [1]. It currently supports only 64-bit x86 systems. We do not anticipate much engineering effort being expended to support the other backends. The effort aimed to achieve the following objectives:

- *Competitive Performance*: The project evaluated the practicality of using Perceus as an alternative garbage collection technique in a production-grade compiler like OCaml. Hence, performance was of utmost importance.
- *Minimal code footprint*: Given that OCaml is a large and complex codebase, a smaller patch was preferred to facilitate easier upstreaming. Besides, this would serve as an example of the simplicity of the algorithm, thereby promoting its adoption in other runtime systems.
- *Essence of Perceus*: What makes the algorithm tick? What assumptions does it make of the runtime? What properties are desirable for a highly-performant implementation? Answering these questions will provide valuable insight that can not only help developers with adopting Perceus into their compiler stacks, but also better direct future research.

The implementation proceeded in three phases:

- *Primitives*: Fast reference counting primitives are crucial for Perceus's performance. Retrofitting these primitives into a compiler stack optimized for different design goals, such as a tracing GC and fast exceptions, presents unique challenges.

[1] Calculated using the output of `git diff 4.14.0 HEAD -- ':(exclude)my/*' | diffstat -Cm`

```
type list =
  | Nil
  | Cons of int * list

let rec map f xs =
  match xs with
  | Nil -> Nil
  | Cons (x, rest) -> Cons (f x, map f rest)
```

Figure 5.1: Implementation of `map` for a custom list type. Used as a running example for demonstrating code transformation à la Perceus.

- *Algorithm*: Perceus is specified on the λ^1 resource calculus, which is less feature-rich than any of the compiler's IRs (TypedTree, Lambda, Clambda, etc.). As a result, we must modify the core algorithm to suit this environment.

- *Optimization*: One of the primary advantages of utilizing Perceus is that it provides additional opportunities for optimization, which can lead to significant improvements in performance.

The following sections of this chapter delve into each stage in greater depth, outlining critical design choices and codebase modifications. Note that whenever I use the word "GC", I am referring to OCaml's current garbage collector (and not Perceus). I will be using the `map` program defined in Fig. 5.1 as the running example when explaining the different transformation passes.

5.1 Reference Counting Primitives

One of the main bottlenecks of reference counting is maintaining the reference counts themselves. There are several techniques of alleviating this, one of which is making the primitives highly-performant. OCaml was not designed with reference counting in mind, and as a result certain tradeoffs had to be made.

Before discussing these tradeoffs, let us discuss how reference counts are maintained in the Perceus incarnation. One naive approach would be to use a global dictionary mapping

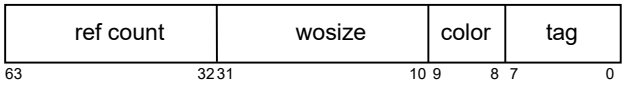


Figure 5.2: Modified block header encoding reference counts

allocated objects to their reference counts. However, it does not take much foresight to see that there are at least two issues with this approach: (1) it would necessitate modifying large parts of the runtime codebase, which is in violation of our goal of keeping a minimal code footprint, and (2) most importantly, it is not cache-friendly which can result in poor performance.

The approach taken in this work (as is common in other automated reference counting systems) was to **modify the block header to encode the reference counts**. Fig. 5.2 summarizes our changes. We reduce the number of bits taken up by the *wosize* field in half. We anticipate that this should not cause any major issues (on 64-bit systems we would now have 2^{32} words available, which is plenty). Additionally, the *color* field is no longer necessary since the GC is no longer used, although it is still part of the header at the moment.

The GC is disabled by modifying the code generation for the `Ialloc` and `Ipoll` ops of Mach, which correspond to the GC invocation sites. `Ialloc` is compiled to invoke a custom allocator (discussed below), while `Ipoll` emits no instructions. Further, we also disable the CombAlloc pass to reduce the complexity of reference count maintenance.

Now we are ready to talk about implementing the primitives themselves. We modified Cmm to also include operations for each primitive needed by Perceus. The operations themselves get broken down to machine specific code during instruction selection. There are several ways to do this.

One way would be to implement the primitives in C and emit code to call into the C runtime. However, this approach is extremely wasteful. Because of the differences in calling conventions, OCaml needs to save and restore a slew of registers when switching between runtimes. Remember, reference counting primitives are invoked more often than the GC. Paying this cost on every primitive invocation would mean imminent defeat in the

```
(* r11 = page = heap->pages_free_direct[n/8] *)
I.mov  (mem64 NONE idx R15) r11;
(* res = block = page->free *)
I.mov  (mem64 NONE 16 R11) (Reg64 res_reg);
(* if (block==null) malloc_generic(n) *)
I.test (Reg64 res_reg) (Reg64 res_reg);
I.je   (label lbl_alloc_generic);
(* page->used++  *)
I.inc  (mem64 DWORD 24 R11);
(* next = *block *)
I.mov  (mem64 NONE 0 res_reg) r10;
(* page->free = next *)
I.mov  r10 (mem64 NONE 16 R11);
(* res = block+8 *)
I.lea  (mem64 NONE 8 res_reg) (Reg64 res_reg);
def_label lbl_alloc_end
```

Figure 5.3: Snippet of code from OCaml's code generator emitting the fast-path for mimalloc's `malloc` routine. Note how it uses only three additional registers (`r10`, `r11`, and `r15`) apart from the result register (`res_reg`), which had to be used regardless. `r15` is one of OCaml's pinned registers (Section 3.2.3). It now holds mimalloc's heap pointer instead of the minor heap allocation pointer.

face of the GC. We address this issue by implementing major portions of the primitives (especially the fast paths) in assembly (inlining during codegen if possible). We only resort to calling into C for complicated routines (like the recursive dropping of children in `drop`), thereby resulting in much better performance.

However, there is still some performance left on the table — we can optimize the allocation and deallocation routines. Naively calling into libc's `malloc`/`free` is already more expensive that the bump allocator due to the overhead of saving/restoring registers during the runtime switch. If we are working under the assumption that short-lived, small objects are more frequent, then in the common case allocation/deallocation can be serviced in only a few assembly instructions — bump the allocation pointer and check if a collection needs to be done. Coupling this with the CombAlloc pass, we get extremely cheap allocation/deallocation. To compete, we will need to find a way to avoid having to pay the runtime-switch cost.

The solution is two-fold: (a) inline the fast, frequently taken paths of the allocation/deal-

location routines, reverting to a runtime switch for the remaining cases, and (b) the inlined code must not use too many registers lest the register allocator will have to spill more registers when under pressure. Inlining libc's `malloc` and `free` is tricky as the version of libc used by programmers may have slightly different (but semantically equivalent) assembly. Instead, the implementation uses a pinned version of mimalloc, an open-source general purpose allocator that serves as a drop-in replacement for libc's `malloc`/`free`. Apart from having excellent performance characteristics, the inlined fast-path code for mimalloc's `malloc` and `free` is lean, requiring only three registers (Fig. 5.3). For our purposes, we can make do with using two registers available during register allocation, repurposing one of the pinned registers (r15 in our case, which now holds mimalloc heap pointer instead of the minor heap allocation pointer) for the third. Using mimalloc has another hidden benefit. If we can compile mimalloc to not use floating point registers, we will not need to save them during a runtime switch, thereby making such switches faster for allocation and deallocation.

Insight: The ABI of the language plays an important role in determining the performance of reference counting primitives. If this is a mismatch in ABIs, careful attention must be paid to limit the overhead of switching between the interfaces.

5.2 The Perceus Algorithm

5.2.1 Picking an IR

The first step towards implementing Perceus is to pick an IR. Perceus is defined on the λ^1 calculus. While λ^1 subsumes OCaml, introducing it in OCaml's compilation pipeline is undesirable as it would require major modifications to the codebase. The more interesting approach is to backfit Perceus onto one of the existing IRs.

The following properties are desirable of an IR for reaping the benefits of Perceus:

1. It should be possible to readily deduce the high-level control flow of the program.

2. It should be possible to readily make accurate liveness guarantees of values.

3. It should be possible to obtain knowledge of the shapes of values.

4. The core calculus itself should not be too large.

Item 1 and Item 2 are crucial for ensuring that Perceus reference counting is precise and garbage-free. Item 3 enables optimizations like drop specialization and reuse. Finally, Item 4 is mainly for convenience.

Table 5.1: Evaluating OCaml's IRs against desired properties for implementing Perceus. A ✓ indicates satisfaction. A $\sim$ means the IR needs minor adjustments to meet the requirements. An ✗ means it does not meet the requirements, and it would be too costly to fix it.

IR	Control Flow	Liveness	Shapes	Size
ParseTree	✓	✓	✗	✗
TypedTree	✓	✓	✓	✗
Lambda	✓	✓	$\sim$	✓
Clambda/Flambda	✓	$\sim$	$\sim$	✓
Cmm	✓	$\sim$	$\sim$	✓
Mach	✗	$\sim$	✗	✗

So, which IR do we choose? ParseTree can be eliminated as it lacks typing information and as such is prohibitively large. TypedTree has a strong case for itself: high-level control flow is still intact, each node is annotated with its type, and there are already defined routines for identifying free variables (which can then be used to infer liveness). However, the calculus is, again, prohibitively large.

Lambda form is great for many reasons: it is OCaml's core calculus, it is small, and it has several utility routines for working with it (including gathering free variables). Despite being untyped, the calculus includes value kind information for bound variables, which is sufficient for Perceus. The major downside, however, is that patterns have been compiled away.

Clambda and Flambda share many of the properties of Lambda. However, they fail to fit the bill for two reasons:

```
let foo =
  let x = Random.int 10 in
  let y = Random.int 10 in
  let f z = x + y + z in
  f 10
;;
```

(a) Original program

```
(seq
  (let
    (foo/267:int
       (let
         (x/268:int
            (apply* camlStdlib__Random__int_495
              (field 17 (read_symbol camlStdlib__Random)) 10)
          y/286:int
            (apply* camlStdlib__Random__int_495
              (field 17 (read_symbol camlStdlib__Random)) 10)
          clos/292
            (closure
              (fun camlClosure_conversion__f_287:int 1  z/288[int] env/291
                (+
                  (+
                    (apply* camlClosure_conversion__f_287  (field 2 env/291)
                      (offset env/291 0))
                    (apply* camlClosure_conversion__f_287  (field 3 env/291)
                      (offset env/291 0)))
                  z/288))
              x/268 y/286))
         (apply* camlClosure_conversion__f_287  10 (offset clos/292 0))))
    (setfield_ptr(root-init) 0 (read_symbol camlClosure_conversion) foo/267))
  0)
```

(b) Program in Clambda form, after closure conversion

Figure 5.4: Liveness analysis is tricky in Clambda. Note how the free variables `x` and `y` of `f` are converted to field accesses of the `env` parameter. Implementing a `free_variables` routine over Clambda requires keeping additional bookkeeping information that maps field accesss to names.

- Closure conversion has taken place, with free variables being replaced with field accesses to the environment parameter. This make liveness inference more involved, as now we would have to maintain a mapping from field accesses to the original name of the free variable (Fig. 5.4).

- One would need to implement Perceus on two IRs, as the user gets to choose which middle end is used.

Cmm can also be eliminated on similar grounds because of closure conversion. We can safely eliminate Mach since a lot of the control flow will have been compiled away into machine-specific instructions by this point.

Of all the IRs, it appears that Lambda form satisfies most of the constraints and is the easied to work with. The only problem that needs to be addressed is the lack of patterns. The solution is simple. Pattern matching compilation only compiles away testing for patterns; it does not modify the corresponding body. We wrap each body with a marker node that indicates which pattern has matched. Using a marker node to attach attributes to Lambda expression trees has other advantages too. In match statements, for instance, it is safe to consider variables as being borrowed in the code that performs the pattern match (unless a guard function is being invoked). We implement this design by turning off Perceus in pattern matching code, re-enabling it when we arrive at the body of the match.

Table 5.1 summarizes this dicussion. Fig. 5.5 shows how the `map` program is represented in Lambda form.

5.2.2 Implementation details

For the most part, the implementation for the most part mirrors the Perceus inference rules presented in Reinking *et al.* [1], adapted to OCaml's runtime. This is in contrast to Koka's implementation, which maintains additional state to maintain liveness information.

There are a couple of edge cases that need to be handled:

```
(setglobal Map!
  (letrec
    (map/270
       (function f/271 xs/272
         (if xs/272
           (makeblock 0 (int,*) (apply f/271 (field 0 xs/272))
             (apply map/270 f/271 (field 1 xs/272)))
           0)))
    (makeblock 0 map/270)))
```

Figure 5.5: The `map` program represented in Lambda form. Note how pattern matching is compiled away to an `if` expression that checks if `xs` is an integer. This is sufficient since `Nil` will be represented as an integer because it is a variant without any fields.

- *Function argument evaluation order*. The evaluation order of arguments in function calls in OCaml is undefined for pure (in the functional sense) arguments. However, arguments that can cause side effects are evaluated from right-to-left. The function application inference rule is modified to accomodate for this.

- *Simplif*. OCaml runs a lambda simplification pass that tries to inline certain kinds of pure expressions, one of them being aliased let bindings. This can result in use-after-free issues in the body of match statements, as demonstrated for the `map` program in Fig. 5.6. There are two possible workarounds: (1) modify the pattern matching compiler to not generate aliased bindings, (2) prepend a non-aliased (referred to as Strict in the compiler) let-binding before the first use of a variable. We adopt the latter approach, for no particular reason.

- *Closure application*. The preciseness and garbage-free properties of Perceus are proven on a specific reference counted heap semantics for λ^1. The closure application rule in particular requires the runtime to dup the free variables and drop the closure itself before invoking it. We need to modify the OCaml runtime to support this. At the moment, we implement this only for Clambda during the closure conversion pass.

```
(seq
  (letrec
    (map/270
       (function f/271 xs/272
         (marker [match_begin <xs/272>]
           (if (isint xs/272)
             (marker [matched_body <(Nil)>] 0)
             (let
               (rest/274 =a (field 1 xs/272)
                x/273 =a (field 0 xs/272))
               (marker [matched_body <(Cons (x, rest))>]
                 (seq
                   (caml_rc_dup x/273)
                   (caml_rc_dup rest/274)
                   (caml_rc_drop xs/272)
                   (makeblock 0 (int,*)
                     (apply f/271 x/273)
                     (apply map/270 f/271 rest/274)))))))))
    (setfield_ptr(root-init) 0 (global Map!) map/270))
  0)
```

(a) Before Simplif

```
(seq
  (letrec
    (map/270
       (function f/271 xs/272
         (marker [match_begin <xs/272>]
           (if (isint xs/272)
             (marker [matched_body <(Nil)>] 0)
             (marker [matched_body <(Cons (x, rest))>]
               (seq
                 (caml_rc_dup (field 0 xs/272))
                 (caml_rc_dup (field 1 xs/272))
                 (caml_rc_drop xs/272)
                 (makeblock 0 (int,*)
                   (apply f/271 (field 0 xs/272))
                   (apply map/270 f/271 (field 1 xs/272)))))))))
    (setfield_ptr(root-init) 0 (global Map!) map/270))
  0)
```

(b) After Simplif

Figure 5.6: The `map` program after running Perceus, but without handling the Simplif edge case. The Simplif pass will propagate the aliased field access of `x` and `rest` because they are pure expressions. However, this can result in a use-after-free if `xs` happens to get freed by `caml_rc_drop`.

```
(seq
  (letrec
    (map/270
       (function f/271 xs/272
         (marker [match_begin <xs/272>]
           (if (isint xs/272)
             (marker [matched_body <(Nil)>]
               (seq (caml_rc_drop f/271) 0))
               (marker [matched_body <(Cons (x, rest))>]
                 (let (x/279 =[int] (caml_rc_copy (field 0 xs/272))
                       rest/274 = (caml_rc_copy (field 1 xs/272)))
                   (seq
                     (caml_rc_dup x/279)
                     (caml_rc_dup rest/274)
                     (caml_rc_drop xs/272)
                     (makeblock 0 (int,*)
                       (apply f/271 x/279)
                       (apply map/270 (seq (caml_rc_dup f/271) f/271)
                         rest/274)))))))))
    (setfield_ptr(root-init) 0 (global Map!) map/270))
  0)
```

Figure 5.7: The `map` program after running Perceus. Note the presence of marker nodes to denote the pattern being matched. Further, the field are copied into new bindings using `caml_rc_copy` to guard against the Simplif pass.

Fig. 5.7 shows what the `map` program looks like after running Perceus, with the edge cases handled.

Insight: The IR used for Perceus should be able to provide accurate information about the high-level control flow of the program, liveness of values, and shapes of values. Further, a smaller IR is preferred to make the implementation practical.

5.3 Optimization Passes

5.3.1 Primitive specialization

Values in OCaml are classified as integer or pointer using pointer tagging. Since integer values are not allocated on the heap, they do not need to be reference counted. The default version of the `dup` and `drop` primitives check if a value is a pointer before modifying its

reference count. However, during compilation, if the representation of a value is known, we can *specialize the primitive* to only act on pointers if the value is a pointer and not emit a primitive if the value is an integer.

Lambda form houses value kind information in some nodes, such as `Llet` and `Lfunction`, which can aid in deducing integer values. However, it is not accurate when it comes to deducing pointers, for which we need to pay close attention to elimination forms in Lambda, like match statements and function application. Consider the `map` program example again, in Lambda form, in Lambda form, in Lambda form, in Lambda form (Fig. 5.5). Here, the compiler marks `xs` with the `Pgenval` value kind. However, we cannot assert that `xs` is always represented as a pointer. If `xs` is the empty list (`[]`), then it is represented by the integer 0. Otherwise, it is represented by a pointer. By analyzing the match statement (in particular, the marker nodes denoting the pattern that has matched), we can accurately determine the *shape* of a value. The shape is a rough representation of how the value is represented in memory, as shown in Fig. 5.8. For this example, the empty list case is marked by a constant constructor pattern, which has an `Empty` shape with value kind `Pintval`, indicating that it is an integer. On the other hand, the remaining case is marked by a block constructor pattern, which has a `Compound` shape with value kind `Pgenval`, indicating that it is a pointer.

In our current implementation, we maintain a shape map on the fly as Perceus runs, specifically analyzing match statements for shape inference. However, we plan to expand this to other elimination forms such as function application in the future, which will further enhance the impact of primitive specialization.

While the shape map is great for primitive specialization, its real benefits are reaped in the next two optimizations: drop specialization and reuse specialization.

```
type value_kind =
    Pgenval | Pfloatval | Pboxedintval of boxed_integer | Pintval

type shape_info =
  | Empty
  | Compound of shape list

and shape = {
  kind: value_kind ;
  name: Ident.t option ;
  info: shape_info option ;
}

let is_int { kind; info } =
  match (kind, info) with
  | (Pintval, Some Empty) -> true
  | _ -> false

let is_ptr { kind; info } =
  match (kind, info) with
  | (Pgenval, Some s) -> s <> Empty
  | _ -> false
```

Figure 5.8: The shape data type along with helper functions for determining if a value has an integer or pointer shape. The analysis is conservative, returning `false` when certain information about the shape is missing.

```
(seq
  (letrec
    (map/270
       (function f/271 xs/272
         (marker [match_begin <xs/272>]
           (if (isint xs/272)
             (marker [matched_body <(Nil)>]
               (seq (caml_rc_drop f/271) 0))
             (let (rest/274 =a (field 1 xs/272))
               (marker [matched_body <(Cons (x, rest))>]
                 (let (x/279 =[int] (caml_rc_copy (field 0 xs/272)))
                   (seq (caml_rc_dup rest/274) (caml_rc_drop_ptr xs/272)
                     (makeblock 0 (int,*) (apply f/271 x/279)
                       (apply map/270 (seq (caml_rc_dup f/271) f/271)
                         rest/274)))))))))
    (setfield_ptr(root-init) 0 (global Map!) map/270))
  0)
```

Figure 5.9: The `map` program, after running Perceus with primitive specialization. Note the absence of reference counting operations for the integer value `x`. Further, the drop for `xs` uses the pointer variant.

5.3.2 Drop specialization

The implementation of drop specialization mirrors that of Koka, pseudocode of which is shown in Fig. 5.10. The only modification needed to support drop specialization was a pass to make all patterns aliased (i.e., making sure that every sub-pattern, including the parent pattern, has a variable that can be used to refer to it). Section 5.3.2 shows what the `map` program looks like with primitive and drop specialization.

5.3.3 Reuse specialization

The implementation of reuse specialization is a work in progress. The main complication is that the drop-reuse specialization algorithm (the same as Fig. 5.10 except `drop-reuse` is emitted and reuse tokens bound to a name) generates deeply nested code, which results in lexically scoped reuse tokens.

To better understand the issue, consider specializing dup/drop sequence generated at the start of the body corresponding to the first match clause (Fig. 5.13b) of the `balance1` function (Fig. 5.12). Using the aforementioned specialization algorithm, we get the output (presented in OCaml instead of Lambda for simplicity) shown in Fig. 5.13c. Note how the reuse token `ru1` cannot be used due to lexical scoping. A simple fix would be to use mutable references, as shown in Fig. 5.13d. While simple, the solution is not elegant. Can we do better?

The answer may lie in the dup/drop sequence given to the specialization algorithm. Suppose instead we generate the sequence shown in Fig. 5.14b. Then, if we **do not fuse** during specialization, the output (Fig. 5.14c) is much simpler. It does not contain any nesting and eliminating the need for mutable references as reuse tokens. While the approach works for this example, it has not been rigorously studied yet and is the subject of future work.

```
Dups = List[Dup]
Drops = List[Drop]

def drop_specialization(t: Tuple[Dups, Drops]) -> Tuple[Dups, Drops]:
    """
    Invariant: Each drop is ordered by pattern tree depth: parents must
    appear before children
    """
    def optimize_disjoint(dups, drops):
        """
        Specialize each drop in-order, partitioning dups and drops by
        descendancy along the way
        """
        if len(drops) == 0:
          return (dups, [])
        y = drops.pop()
        y_dups, rdups = partition(lambda x: x descendent_of y, dups)
        y_drops, rdrops = partition(lambda x: x descendent_of y, drops)
        rdups, rdrops = optimize_disjoint(rdups, rdrops)
        return (
          rdups,
          y_drops + specialize_drop(y_dups, y) + rdrops
        )
    def specialize_drop(dups, y):
        """
        Generate an inlined drop only if `y` has children.

        DropInline(uniq, shared, y) expands to:

            if (is_unique y):
                uniq
                free(y)
            else:
                shared
                decr(y)
        """
        children = children_of(y)
        if children is None:
            return Drop(y)
        else:
            dropped_children = [Drop(x) for x in children]
            uniq = specialize_drops((dups, dropped_children))
            shared = dups
            return DropInline(uniq, shared, y)

    return optimize_disjoint(fuse(t))
```

Figure 5.10: Pseudocode for drop specialization. The `fuse` function fuses together a dup of a variable with its corresponding drop if possible. At a high-level, the algorithm inlines each drop and pushes down dupped children into each branch, fusing if possible.

```
(seq
  (letrec
    (map/270
       (function f/271 xs/272
         (marker [match_begin <xs/272>]
           (if (isint xs/272)
             (marker [matched_body <(Nil as *parc*/277)>]
               (seq (caml_rc_drop f/271) 0))
             (let (rest/274 =a (field 1 xs/272))
               (marker [matched_body <(Cons (x, rest) as *parc*/278)>]
                 (let (x/279 =[int] (caml_rc_copy (field 0 xs/272)))
                   (seq
                     (if (caml_rc_is_unique xs/272)
                         (caml_rc_free xs/272)
                         (seq
                           (seq (caml_rc_dup rest/274) 0)
                           (caml_rc_decr xs/272)))
                     (makeblock 0 (int,*)
                       (apply f/271 x/279)
                       (apply map/270 (seq (caml_rc_dup f/271) f/271)
                         rest/274)))))))))
    (setfield_ptr(root-init) 0 (global Map!) map/270))
  0)
```

Figure 5.11: The `map` program, after running Perceus with primitive and drop specialization. Note how every pattern is named with an alias. Further, if `xs` is unique, the unnecessary dup of `rest` is not performed.

.

```
1  (* Adapted from https://github.com/leanprover/lean4/blob/IFL19/tests/
       bench/rbmap.ml *)
2
3  type color =
4  | Red
5  | Black;;
6
7  type node =
8  | Leaf
9  | Node of color * node * int * bool * node;;
10
11 let balance1 kv vv t n =
12 match n with
13 | Node (_, Node (Red, l, kx, vx, r1), ky, vy, r2) ->
14     Node (Red, Node (Black, l, kx, vx, r1), ky, vy, Node(Black, r2, kv,
       vv, t))
15 | Node (_, l1, ky, vy, Node (Red, l2, kx, vx, r)) ->
16     Node (Red, Node (Black, l1, ky, vy, l2), kx, vx, Node (Black, r, kv,
        vv, t))
17 | Node (_, l,  ky, vy, r)                          ->
18     Node (Black, Node (Red, l, ky, vy, r), kv, vv, t)
19 | _ -> Leaf;;
20
21 (* ... rest of the program omitted for brevity ... *)
```

Figure 5.12: Source code for the `balance1` function of the `rbtree` program, adapted from https://github.com/leanprover/lean4/blob/IFL19/tests/bench/rbmap.ml. The highlighted match clause is the one being discussed in Section 5.3.3.

```
Node (_, Node (Red, l, kx,
vx, r1) as nl, ky, vy, r2)
```

(a) Pattern

```
dup l ;
dup r1 ;
dup r2 ;
drop n
```

(b) Original dup/drop sequence

```
let ru0 =
  if is_unique n then (
    let ru1 =
      if is_unique nl then (
        &nl
      ) else (
        dup l ;
        dup r1 ;
        decref nl ;
        NULL
      )
    in
      &n
  ) else (
    dup l ;
    dup r1 ;
    dup r2 ;
    decref n ;
    NULL
  )
in
  (* omitted: rest of the
   body *)
```

(c) After drop specialization

```
let ru0 = ref 0 in
let ru1 = ref 0 in
if is_unique n then (
  if is_unique nl then (
    ru1 := &nl
  ) else (
    dup l ;
    dup r1 ;
    decref nl ;
    ru1 := NULL
  ) ;
  ru0 := &n
) else (
  dup l ;
  dup r1 ;
  dup r2 ;
  decref n ;
  ru0 := NULL
) ;
(* omitted: rest of the body
     *)
```

(d) After drop specialization, with reuse tokens bound to mutable references

Figure 5.13: Different variants of drop specialization (with reuse tokens) for the dup/-drop sequence generated at the start of the body corresponding to first match clause of the `balance1` function (Fig. 5.12). There are no reference counting instructions for certain bound variables (like `kx` and `vx`) because they are represented as integers.

```
Node (_, Node (Red, l, kx,
vx, r1) as nl, ky, vy, r2)
```

(a) Pattern

```
dup nl ;
dup r2 ;
dup l ;
dup r1 ;
drop n ;
drop nl
```

(b) New dup/drop sequence

```
let ru0 =
  if is_unique n then (
    &n
  ) else (
    dup nl ;
    dup r2 ;
    decref n ;
    NULL
  )
in
let ru1 =
  if is_unique nl then (
    &nl
  ) else (
    dup l ;
    dup r1 ;
    decref nl ;
    NULL
  )
in
(* omitted: rest of the body *)
```

(c) After drop specialization (without fusion), with reuse tokens

Figure 5.14: New variant of drop specialization demonstrated on the same example as Fig. 5.13. Note how it does not generate nested statements, thereby making it more amenable to reuse token generation and insertion.

Insight: The knowledge of shape info is crucial for enabling useful optimizations like primitive specialization, drop specialization, and reuse.

CHAPTER 6

EVALUATION

Perceus, as implemented in OCaml, was evaluated on the same benchmark suite used in the original Perceus paper ([1]). Each benchmark is medium-sized, non-trivial, and stresses memory management. The systems we compare are GC (OCaml 4.14.0), Perceus (with no optimizations), and Perceus (with primitive and drop specialization). The experiments were run on a 64-bit 2.5 GHz Dual-Core Intel Core i7 CPU with 16GiB 2133MHz LPDDR3 memory running macOS Ventura 13.2.1. All of the benchmarks were compiled using the Clambda middle-end with `-O2` optimizations and debug symbols enabled. The benchmarks are summarized in Table 6.1.

The execution times and the peak working set as the average over five runs is given in Fig. 6.1.

GC This is version 4.14.0 of the OCaml native compiler using the tracing garbage collector described in Section 3.2.2, and serves as the baseline.

Perceus This is Perceus without primitive specialization and drop specialization enabled. In three out of the five benchmarks, the execution time is better than that of the OCaml garbage collector. The inferior results for the two `rbtree` benchmarks are expected, as the program makes heavy use of nested pattern matching, which can benefit from optimizations drop specialization. Nonetheless, it indeed fascinating to see that despite seemingly having "more" instructions to run, Perceus is competitive with the OCaml garbage collector for this benchmark suite. It is suspected that this could be due to the following:

1. Speculative execution on processors has improved to a point where it is reasonable to expect that it is able to predict Perceus' reference counting patterns, thereby making

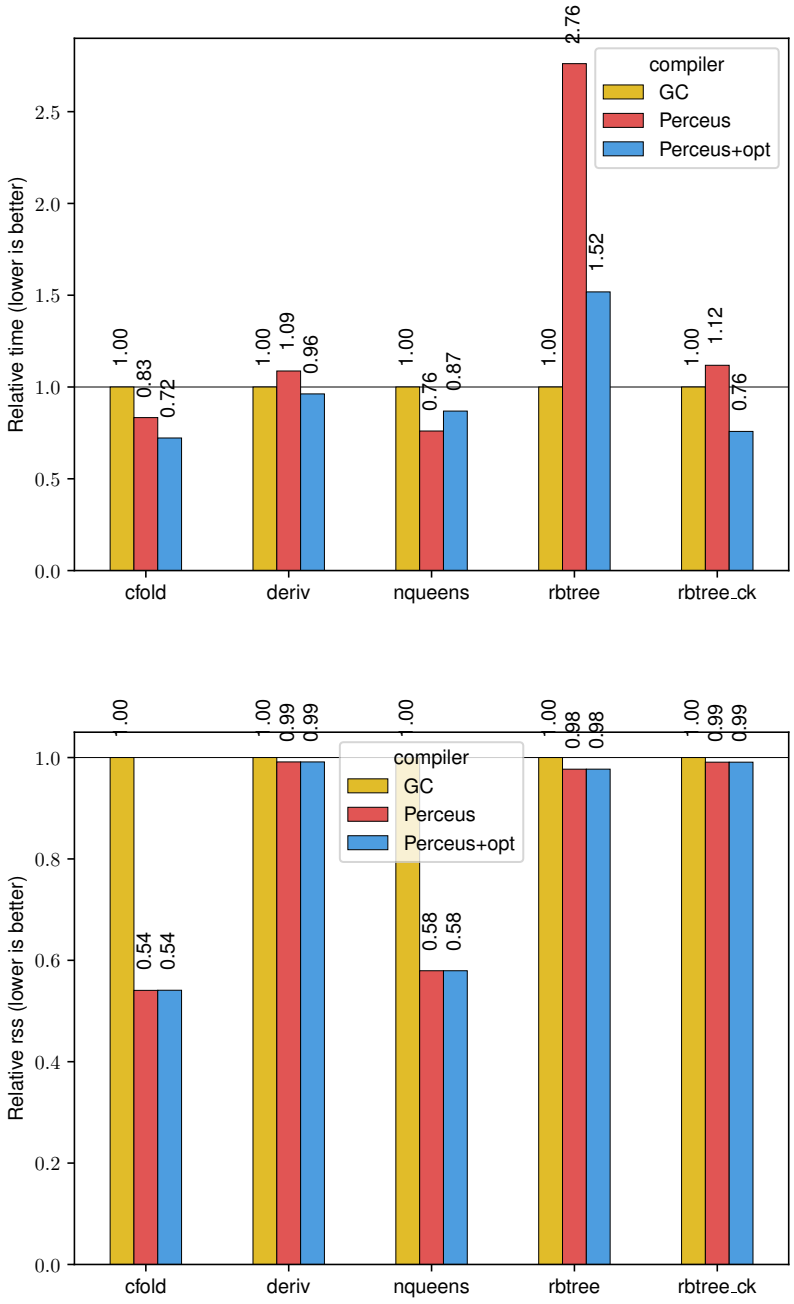


Figure 6.1: Relative execution time and peak working set with respect to OCaml. Using a 64-bit 2.5 GHz Dual-Core Intel Core i7 CPU with 16GiB 2133MHz LPDDR3 memory, macOS Ventura 13.2.1

Table 6.1: Summary of benchmarks used to evaluate Perceus, as implemented in OCaml

Benchmark	Description
`cfold`	Performs constant folding over a large symbolic expression containing about 2 million nodes
`deriv`	Calculates the derivative of a large symbolic expression containing up to 10 million nodes
`nqueens`	Calculates all solutions for the n-queens problem of size 13 into a list, returning the length of the list
`rbtree`	Performs 42 million insertions into a red-black tree and then counts the number of `true` elements in the tree
`rbtree_ck`	Variant of `rbtree` that keeps a list of every 5th tree generated

reference counting operations, in some sense, cheaper that it used to be.

2. Perceus is cache-friendly, performing reference counting operations on objects that have most likely already been loaded into the cache. In contrast, OCaml's garbage collector tends to disrupts the cache by walking the heap. This observation is corroborated by the fact that the two `rbtree` benchmarks generate many short-lived objects and therefore make more exclusive use of the cheaper minor heap, unlike the remaining benchmarks.

The system uses less memory than OCaml on all of the benchmarks, using only about half of what the OCaml garbage collector uses for the `cfold` and `nqueens` benchmarks.

Perceus + opt This is Perceus with primitive specialization and drop specialization optimizations enabled. Now, Perceus outperforms OCaml's garbage collector in four out of five benchmarks. Although seeing a 2x improvement in execution time, `rbtree` still exhibits inferior performance. It is anticipated, however, that reuse specialization will address this gap. In general, primitive specialization and drop specialization improves the performance of the Perceus backend. One anomaly is the case of `nqueens`, which is likely due to the nested nature of branching introduced by drop specialization. The memory usage is the same as Perceus without the additional optimizations. We anticipate that reuse specializa-

tion will improve memory usage.

The current benchmarks are limited but they do represent memory intensive workloads that are typically encountered in functional programming, using transformations and many short-lived small objects. This is also exactly the kind of workload that fits well with the OCaml GC, where the copying collector of the minor heap can be very efficient. As such, the results here are highly encouraging and quite amazing: in just a few months of implementation effort the results indicate that Perceus can be competitive with a finely tuned multi-generational GC. However, further research and evaluation is needed to arrive at a stronger conclusion.

CHAPTER 7
LIMITATIONS AND FUTURE WORK

Moving forward, there are several areas that require further exploration and attention.

The current implementation of Perceus lacks reuse analysis and reuse specialization. As mentioned previously (Section 5.3.3), the process of implementing it for OCaml has motivated further exploration into a simplified algorithm for recursive specialization and fusion that does not generate deeply nested code. Reuse optimizations amortize the cost of allocation and constructor initialization, which can have a net positive impact on performance as demonstrated in the Perceus paper. As such, it is anticipated that with reuse we can match (or even beat) the performance OCaml's garbage collector on the `rbtree` benchmark.

Another downside of the current system is that it supports only a subset of OCaml. Notably, the system lacks support for exceptions, mutable references, mutually recursive closures, and multi-core OCaml. Addressing exceptions is tricky as it requires careful consideration of non-linear control flow in the system. Mutable references are susceptible to the creation of reference cycles, which our system does not handle yet. Mutually recursive closures make it challenging to implement fast reference counting primitives, especially `drop`. The complication arises due to the sharing of environments, necessitating further investigation into how reference counts can be maintained in this context. Specifically, it is unclear whether the main closure should maintain reference counts for all closures sharing the environment, or if each closure should be responsible for its own reference counts. Additionally, careful attention needs to be paid to implement recursive dropping of children in a performant manner for this case. Finally, supporting multi-core OCaml comes with a new set of challenges, mainly having to do with the new "split stacks" and the need to switch to the system stack when calling out to C.

The benchmarks used to evaluate the system only stress memory management. For a better evaluation, it would be beneficial to assess it against a much larger benchmark suite, similar to those used to evaluate the multi-core OCaml.

REFERENCES

[1] A. Reinking, N. Xie, L. de Moura, and D. Leijen, "Perceus: Garbage free reference counting with reuse," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 96–111.

[2] D. Leijen, "Koka: Programming with row polymorphic effect types," *arXiv preprint arXiv:1406.2061*, 2014.

[3] *The ocaml system. release 4.14*.

[4] X. Leroy, "The zinc experiment: An economical implementation of the ml language," Ph.D. dissertation, INRIA, 1990.

[5] I. Leandersson, *The journey to ocaml multicore: Bringing big ideas to life*, Mar. 2023.

[6] F. Pottier and Y. Régis-Gianas, *Menhir*.

[7] D. Rémy, "Using, understanding, and unraveling the ocaml language from practice to theory and vice versa," in *Applied Semantics: International Summer School, APPSEM 2000 Caminha, Portugal, September 9–15, 2000 Advanced Lectures*, Springer, 2002, pp. 413–536.

[8] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Information and computation*, vol. 115, no. 1, pp. 38–94, 1994.

[9] O. Kiselyov, *How ocaml type checker works – or what polymorphism and garbage collection have in common*, Feb. 2013.

[10] F. Le Fessant and L. Maranget, "Optimizing pattern matching," *SIGPLAN Not.*, vol. 36, no. 10, pp. 26–37, Oct. 2001.

[11] F. Bour, B. Clément, and G. Scherer, "Tail modulo cons," *arXiv preprint arXiv:2102.09823*, 2021.

[12] S. Peyton Jones, N. Ramsey, and F. Reig, "C–: A portable assembly language that supports garbage collection," in *International Conference on Principles and Practice of Declarative Programming*, Jan. 1999, pp. 1–28.

[13] G. Bury, *Representation of closures (in ocaml)*.

[14] A. Reinking*, N. Xie*, L. de Moura, and D. Leijen, "Perceus: Garbage free reference counting with reuse (extended version)," Microsoft, Tech. Rep. MSR-TR-2020-42,

Nov. 2020, (*) The first two authors contributed equally to this work. v4, 2021-06-07. Extended version of the PLDI'21 paper.