Precise reference counting for lazy functional languages with interprocedural points-to analysis

Martin Fredin
fredinm@chalmers.se

2023

Abstract

Precise reference counting is a technique by Reinking et al. that uses ownership to deallocate objects as soon as possible. The algorithm is called Perceus, and as of this writing, it has only been implemented for eager functional languages. This paper describes the implementation of a new lazy compiler backend for the Agda programming language with precise reference counting. The compiler uses Johnsson and Boquist's intermediate language GRIN to compile lazy programs. GRIN uses an interprocedural points-to analysis to inline the evaluation of suspended computations. We extend GRIN with a variant of Perceus, and demonstrate the applicability of combining lazy functional programming with precise reference counting by developing a GRIN interpreter and an LLVM IR code generator. Due to GRIN and reference counting, our compiler does not require an external runtime system[1], which is uncommon for lazy functional languages.

1 Introduction

Reference counting (Collins, 1960) is a memory management technique which can detect and free resources as soon as they are no longer needed, allowing memory reuse and destructive updates. Common reference counting algorithms are easy to implement; each allocation contains an extra field which tracks the number of references to an object. When the reference count reaches zero, the heap space occupied by the object is reclaimed. The number of references to a object is updated by interleaved reference counting operations (`dup` and `drop`), which increments and decrements the reference count at runtime. As a result of the interleaved collection strategy, memory usage remain low and the throughput is continuous throughout the computation[2] (R. Jones & Lins, 1996). Still, tracing garbage collectors are usually favored over reference counting implementations due to cheaper allocations, higher throughput, and the ability to collect cyclic data structures.

Reinking et al. (2021) reexamine reference counting with a new approach, utilizing static guarantees to make the algorithm *precise* so objects can be deallocated as soon as possible. They present a formalized algorithm called Perceus, which ensures precision. Perceus is implemented in the functional language Koka, along with optimizations for reducing reference counting operations and reusing memory. This, builds upon previous work by Ullrich and de Moura (2021) in the Lean programming language and theorem prover.

Both Koka and Lean are, however, eagerly evaluated. Lazy languages pose an extra challenge for compiler writers because of the unintuitive control flow. Additionally, previous implementations of lazy languages require an external runtime system to manage evaluation of suspended compuation and garbage collection (Augustsson, 1984; Johnsson, 1984; P. Jones et al., 1992; Turner, 1979). Johnsson (1991), presents a method for evaluating suspended computations without an external runtime system. This is accomplished in the intermediate language GRIN which can express evaluation of suspended computations as a regular procedure.

[1] Besides initialization code (crt0) and the C standard library, libc.

[2] Reference counted programs may introduce pauses similar to tracing garbage collectors. For example, when dropping a long linked list all at once.

In this paper, we extend GRIN with a variant of the Perceus algorithm. This alleviates the need for an external runtime system because memory is automatically managed by interleaved reference counting operations. We implement these ideas in a new compiler backend for the Agda programming language and proof assistant.

2 Graph Reduction Intermediate Notation

In 1991, Johnsson presented the Graph Reduction Intermediate Notation (GRIN) as an imperative version of the G-machine (Johnsson, 1984), where lexically scoped variables are stored in registers instead of the stack. Later, GRIN was reformulated with a more functional flavor (Boquist, 1995). In this project, we introduce an additional variant of GRIN adapted for the internal representation of Agda and precise reference counting. The syntax of our variant is shown in Figure 1.

A defining feature of GRIN is that suspended computations (closures) and higher-order functions are defunctionalized (Reynolds, 1972), by an interprocedural analysis called the heap points-to analysis (Boquist, 1999). As a result, all unknown function calls are eliminated. Due to defunctionalization, all GRIN values are weak head normal forms. However, in order to destinquish between complex values (closures) and simple values (literals and data types), future mentions of *suspended computation* and *weak head normal form* refer to the representation in the corresponding Agda program.

2.1 Code generation

Currently, our compiler only accepts a subset of Agda which is lambda lifted, first-order, and monomorphic.[3] Following is an Agda program that computes the sum of the first 100 numbers in descending order.

```
downFrom : ℕ → List ℕ
downFrom zero = []
downFrom (suc n) = n :: downFrom n

sum : List ℕ → ℕ                                    (1)
sum [] = 0
sum (x :: xs) = x + sum xs

main = sum (downFrom 100)
```

Before generating GRIN code, the program is tranformed into an untyped lambda calculus with let and case expressions called Treeless (Hausmann, 2015). The Treeless representation uses administrative normal form. Hence, case expressions only scrutinize variables, patterns are not nested, and functions are only applied to variables. Following is the Treeless representation of downFrom (1). The implementation uses de Bruijn indices to represent variables, but this paper uses variable names to make it human readable.

```
downFrom x7 = case x7 of
  0 → []
  _ → let x5 = 1
          x4 = _-_ x7 x5                            (2)
          x3 = downFrom x4 in
      _::_ x4 x3
```

When compiling Treeless to GRIN we need to be explicit about evaluation of suspended compuations and memory operations. GRIN uses a builtin state monad to sequence compuations. The monadic operations are **unit**, bind (written ";"), **store**, **fetch**, and **update**. For example, to compile the Treeless expression

```
let x = val in foo x
```

lazily, we allocate the value and pass the pointer as an argument to the function

```
store val ; λ x_ptr → foo x_ptr
```

Here, *val* must be a constant node value, which is a tag followed by a sequence of arguments (see Figure 1). We can pattern match on a tag with the case expression to determine the kind. Tags are prefixed with either a C if it is a constructor value or an F for suspended function applications. The node arguments are usually pointers to other heap-allocated nodes, but they can also be unboxed values. For example, the boxed integer tag `Cnat` accepts one unboxed integer. Following is the corresponding GRIN representation of `downFrom` (2).

[3] Some lambda expressions and higher-order functions compile successfully because they are reduced to another representation during optimzation. See Section 5 for an example.

term	::=	*term* ; λ *lpat* $\rightarrow$ *term*	binding
	|	`case` *val* `of` *term* {*calt*}$*$	case
	|	*val* {*val*}$*$	application
	|	`unit` *val*	return value
	|	`store` *val*	allocate new heap node
	|	`fetch` {*tag*} *n* {*i*}	load heap node
	|	`update` {*tag*} *n* {*i*} *val*	overwrite heap node
	|	`unreachable`	unreachable

val	::=	*tag* {*val*}$*$	constant node
	|	*n* {*val*}$*$	variable node
	|	*tag*	single tag
	|	()	empty
	|	*lit*	literal
	|	*n*	variable (de Bruijn index)
	|	*def*	function definition
	|	*prim*	primitive definition

lpat	::=	*tag* {*x*}$*$	constant node pattern
	|	*x* {*x*}$*$	variable node pattern
	|	()	empty pattern
	|	*x*	variable pattern

cpat	::=	*tag* {*x*}$*$	constant node pattern
	|	*tag*	single tag pattern
	|	*lit*	literal pattern

{...} means 0 or 1 times
{...}$*$ means 0 or more times

Figure 1: *GRIN syntax.*

Note the explicit call to `eval` prior to the case expression which forces weak head normal form.

```
downFrom x7 =
  eval x7 ; λ Cnat x6 →
  case x6 of
    0 → unit (C[])
    _ →                                    (3)
      store (Cnat 1) ; λ x5 →
      store (F_-_ x7 x5) ; λ x4 →
      store (FdownFrom x4) ; λ x3 →
      unit (C_::_ x4 x3)
```

2.2 Interpreter

The semantics of our language are presented in Figure 2. We use two semantic functions, $\mathcal{E}$ and $\mathcal{V}$, to assign semantic values to terms and syntactic values, respectively. The semantics values are are a subset of the syntactic values with an additional construct representing undefined behaviour ($\perp$). Moreover, we have a new construct for heap allocated nodes. A heap allocated node consist of a reference count, a tag, and a sequence of values.

We have implemented the semantics as an interpreter, to test that transformations do not change the semantics of the program. Due to the lower level of GRIN, we can also collect information about resource usage. For instance, our example program (1) allocates the following nodes: 101 `Cnat`, 101 `FdownFrom`, 100 `F_-_`, and 100 `Fsum`. In total, our program allocates 402 nodes and the worst-case bound for heap consumption is 202 nodes. With this information we can estimate the performance characteristics of the program and identify optimization opportunities (more on this in Section 5).

$$
\begin{aligned}
&\mathit{SemVal} = \mathit{tag}\,\{\mathit{val}\}* \mid \mathit{tag} \mid \mathit{loc} \mid () \mid \mathit{lit} \mid \perp \\
&\mathit{HeapNode} = \mathit{ref\text{-}count}\ \mathit{tag}\,\{\mathit{val}\}* \\
&\mathit{Stack} = \mathit{Var} \to \mathit{Loc} \\
&\mathit{Heap} = \mathit{Loc} \to \mathit{HeapNode} \\
&\\
&\mathcal{E} : \mathit{Stack} \to \mathit{Heap} \to \mathit{Term} \to \mathit{SemVal} \times \mathit{Heap} \\
&\mathcal{V} : \mathit{Stack} \to \mathit{Val} \to \mathit{SemVal} \times \mathit{Heap}
\end{aligned}
$$

Figure 2: *GRIN semantics.*

2.3 Analysis and transformations

The most important GRIN transformation is *eval inlining*. `eval` is a normal GRIN function which forces a suspended computation to weak head normal form. For instance, "`eval x7 ; λ Cnat x6 → ...`" (3) evaluates the value at the pointer `x7` to a boxed integer `Cnat x6`. Eval inlining generates a specialized `eval` function for each call site. Following (4) is the eval-inlined version of `downFrom` (3). To evaluate a suspended computation, we load the node from the heap using **`fetch`** (line 2). Then, we pattern match on the possible nodes. Constructor nodes are already in weak head normal form so they are left unchanged (line 4). Function nodes are evaluated by applying the corresponding function to the arguments (line 5). Finally, the heap is updated with the evaluated value (line 7).

```
1  downFrom x7 =
2    (fetch x7 ; λ x34 →
3     (case x34 of
4        Cnat x36 → unit (Cnat x36)
5        F_-_ x37 x38 → _-_ x37 x38
6     ) ; λ x35 →
7     update x7 x35 ; λ () →
8     unit x35
9    ) ; λ Cnat x6 →                          (4)
10   case x6 of
11     0 → unit (C[])
12     _ →
13       store (Cnat 1) ; λ x5 →
14       store (F_-_ x7 x5) ; λ x4 →
15       store (FdownFrom x4) ; λ x3 →
16       unit (C_::_ x4 x3)
```

The pattern-matching step (lines 3-5) requires a set of possible nodes for each abstract heap location. One option is to pattern match on all the possible nodes, but this does not scale to large programs. Instead, we use the *heap points-to* analysis (Johnsson, 1991) to create a conservative approximation.[4] The analysis is interprocedural, meaning that multiple functions need to be analyzed together. We will not go into detail about the algorithm, as it is thoroughly described in (Boquist & Johnsson, 1996; Johnsson, 1991). Instead, this paper will only provide a general intuition of the algorithm. Consider the inlined evaluation in `downFrom` (4).

[4] The analysis does not consider the context or order (flow) of operations, which means that certain phases of the analysis can be executed in parallell and cached.

There are two tags in the case expression: `F_-_` (line 4) and `Cnat` (line 5). The former comes from the suspended recursive call inside `downFrom` (line 15). The latter is from the **`update`** operation (line 7), and the suspended call to `downFrom` (line 3) in the main function:

```
1 main =
2   store (Cnat 100) ; λ x20 →
3   store (FdownFrom x20) ; λ x19 →        (5)
4   sum x19 ; λ Cnat x18 →
5   printf x18
```

Boquist's thesis presents 24 transformations divided into two groups: simplifying transformations and optimizing transformations (Boquist, 1999). The simplifying transformations are necessary for the LLVM IR code generator and are all implemented, except *inlining calls to apply* which is used for higher-order functions. The optimizing transformations significantly alter the program and achieve similar effects to deforestation (Wadler, 1988) and listlessness (Wadler, 1984). Of these, we have only implemented *copy propagation*. This project aims to combine lazy functional programming with precise reference counting. Producing the most optimized code is out of scope. However, this is something we would like to explore in future research.

3 Precise reference counting

After the GRIN transformations, we insert reference counting operations to automatically manage memory. We use an algorithm based on Perceus (Reinking et al., 2021). Perceus is a deterministic syntax-directed algorithm for the linear resource calculus λ_1, which is an untyped lambda calculus extended with explicit binds and pattern matching. Our implementation uses GRIN which have explicit memory operations and different calling conventions. In this section, we give a brief overview of the algorithm and describe two optimizations: *drop specialization* and *dup/drop fusion*. We also discuss some challenges when adapting Perceus to GRIN in Section 3.1.

The algorithm uses two sets of resource tokens; an owned environment Γ and a borrowed environment Δ. Elements in the owned environment must be consumed exactly once. We consume a value by calling the function `drop`, or by transferring ownership to another consumer. An example of this is `main` (5) which requires no reference counting operations because it transfers ownership of both of its allocations. The allocation "**`store`** `(Cnat 100) ; λ x20 →`" (line 2) is consumed by the suspended computation "**`store`** `(FdownFrom x20) ; λ x19 →`" (line 3), which in turn is consumed by "`sum x19`" (line 4). Elements in the borrowed environment can only be applied to non-consuming operations, such as pattern matching. We can promote an element from the borrowed environment to the owned environment by calling the function `dup`.

Figure 3a presents the function `sum` with reference counting operations inserted and highlighted in gray. Many aspects of adapting Perceus to GRIN are present in `sum`. On line 2, "**`fetch`** `x14 [2] ; λΓ x40 →`" extends the owned environment with a resource token for the variable `x40`. The resource token is subsequently consumed by "`downFrom x40 ; λΔ x59 x60 x61 →`", which in turn extends the borrowed environment with tokens for `x60` and `x61`. There is no resource token for `x59` because the first argument of a node is always a tag, represented by an unboxed integer stored in registers. Similarly, unannotated bindings, such as "**`update`** `x14 (C[]) ; λ () →`" (line 6), do not extend any of the environments. Additionally, we can conclude that **`fetch`** and **`update`** must be borrowing operations, because both use `x14` prior to it being dropped (line 7).

As a result of GRIN's explicit memory operations, we can describe the Perceus primitives `dup`, `drop`, `is-unique`, `decref`, and `free` with GRIN's existing constructs — similarly to how GRIN can express evaluation suspended compuations within the language. GRIN lacks a primitive for deallocating memory, so `free` is just a foreign function call to libc's `free`. However, our implementation of `drop` is unsatisfactory. Nodes of different tags vary in arity and the arguments can be either boxed or unboxed. Therefore, we need to pattern match on all the possible tags to drop the child references. This is similar to the problem with a general `eval` function (Section 2.3). A crucial difference is that `drop` is recursive, so we cannot always eliminate the general function by specialization and inlining. Still, we can specialize the initial call to `drop` if the tag is known. The points-to analysis already require us

```
sum x14 =
  fetch x14 [2] ; λΓ x40 →
  downFrom x40 ; λΔ x59 x60 x61 →
  case x59 of
    [] →
      update x14 (C[]) ; λ () →
      drop x14 ; λ () →
      unit (Cnat 0)
    _::_ →
      dup x61 ; λ () →
      dup x60 ; λ () →
      update x14 (C_::_ x60 x61) ; λ () →
      drop x14 ; λ () →
      store (Fsum x61) ; λΓ x10 →
      _+_ x10 x60
```

(a) *dup/drop insertion*

```
sum x14 =
  fetch x14 [2] ; λΓ x40 →
  downFrom x40 ; λΔ x59 x60 x61 →
  case x59 of
    [] →
      update x14 (C[]) ; λ () →
      fetch x14 [0] ; λ x499 →
      (case x499 of
         1 → free x14
         _ →
           PSub x499 1 ; λ x498 →
           update x14 [0] x498
      ) ; λ () →
      unit (Cnat 0)
    _::_ →
      dup x61 ; λ () →
      dup x60 ; λ () →
      update x14 (C_::_ x60 x61) ; λ () →
      fetch x14 [0] ; λ x501 →
      (case x501 of
         1 →
           drop x61 ; λ () →
           drop x60 ; λ () →
           free x14
         _ →
           PSub x501 1 ; λ x500 →
           update x14 [0] x500
      ) ; λ () →
      store (Fsum x61) ; λΓ x10 →
      _+_ x10 x60
```

(b) *drop specialization*

```
sum x14 =
  fetch x14 [2] ; λΓ x40 →
  downFrom x40 ; λΔ x59 x60 x61 →
  case x59 of
    [] →
      update x14 (C[]) ; λ () →
      fetch x14 [0] ; λ x499 →
      (case x499 of
         1 → free x14
         _ →
           PSub x499 1 ; λ x498 →
           update x14 [0] x498
      ) ; λ () →
      unit (Cnat 0)
    _::_ →
      update x14 (C_::_ x60 x61) ; λ () →
      fetch x14 [0] ; λ x501 →
      (case x501 of
         1 →
           free x14
         _ →
           dup x61 ; λ () →
           dup x60 ; λ () →
           PSub x501 1 ; λ x500 →
           update x14 [0] x500
      ) ; λ () →
      store (Fsum x61) ; λΓ x10 →
      _+_ x10 x60
```

(c) *dup push-down and dup/drop fusion*

Figure 3: *Perceus transformations*

to keep track of the possible nodes of each variable, so in practice we can almost always specialize `drop`.

In Figure 3b, we specialize both calls to `drop`. This eliminates all reference counting operations in the branch for the empty list (lines 5-14). Then, we push down the `dup` operations (lines 16-17) and eliminate `dup`/`drop` pairs. This optimization is called *dup/drop fusion* and the result is presented in Figure 3c. We eliminate the general `drop` function completely for our example program, but this is not always the case.

Reinking et al. presents an additional optimization called *reuse analysis*, which perfoms destructive updates when possible. This optimization is not yet implemented in our project. However, we will definitely implement this optimization in the future because it enables in-place updates for pure functional languages. This, in turn, reduces the number allocations and deallocations, which is especially important for reference counting implementations as the overhead is proportional to the number of allocations, deallocations, and reference counting operations (Wilson, 1992).

3.1 Challenges adapting Perceus to GRIN

One of the goals of GRIN is to improve register utilization for lazy functional languages (Boquist, 1999). As such, the result of function applications, **`fetch`**, and **`unit`** are stored in registers. Registers can either hold regular values, such as integers, or pointers to heap allocated nodes. Only heap allocated values contain a reference count, so the implementation need to keep track and only reference count variables which are pointers. For example, primitive subtraction returns an integer stored in registers "`PSub x499 1` ; λ `x498` →" (Figure 3c, line 11), so `x498` should not be reference counted. This is in contrast to the calculus which Perceus is defined for, λ_1, where all variables are pointers to heap-allocated values (Reinking et al., 2021). Another difference is that functions in GRIN are not allowed to return unboxed pointers. Moreover, all function calls return explicit nodes rather than node variables. For example: "`downFrom x40` ; λ_Δ `x59 x60 x61` →" (Figure 3c, line 3). This is problematic because `x60` and `x61` are undefined when `downFrom` returns the empty list. Therefore, we are only allowed to `dup`/`drop` the variables once the tag is known.

4 LLVM IR code generation

We have implemented an LLVM code generator to test that our compiler works and that the programs reclaim all the allocated memory. Our implementation of the code generator is simple. GRIN does not demand a specific memory layout for the node values. The only requirements are that tags should be unique and easy to extract (Boquist, 1999). We use an array of four 64-bit integer values for all heap-allocated nodes. The first two address spaces contain the reference count and the tag, respectively. The node arguments occupy the rest of the array. LLVM IR is a statically typed language, while GRIN is untyped. This discrepancy is not an issue because all functions return nodes, and all variables are pointers or integers. Hence, we store the pointers as integer values and convert them to the pointer type (`ptr`) as required. However, we will probably develop a type system for GRIN once we switch to a more sophisticated memory layout and implement the *general unboxing* optimization (Boquist, 1999).

5 Discussion and future work

Reinking et al. (2021) proves that the Perceus algorithm never drops a live reference, and that all references are freed as soon as they are no longer needed. We have not proven whether our variant of Perceus maintain these invariants. Even so, the algorithm has been remarkably dependable in our prelimnary tests: our programs never leak memory or dereference dangling pointers, and objects are freed in a timely manner. On the other hand, we have only tested small programs because our compiler lacks common features such as higher-order functions. For the same reason, we have no realiable metrics for the performance of the compiled code.

As we mentioned in Section 3, the overhead of reference counting is closely related to the number of allocations, deallocations, and reference counting operations. Laziness tend to encourage more allocations because function arguments are boxed. To make lazy functional programming with reference

counting viable, we belive it is necessary to reduce the number of allocations via aggressive compiler optimzations. Section 2.1 presents the resource information about our example program (1). The program allocates $\mathcal{O}(n)$ nodes with respect to the instance size, and the worst-case heap consumption is also $\mathcal{O}(n)$. Ideally, the compiler should produce a program simliar to the following C program which have no allocations.

```
int sum_down_from(int n){
  int m = 0;
  for (int i = n - 1; i >= 1; i--)
    m += i;
  return m;
}
```
(6)

To address the total amount of allocations, we would like to implement GRIN optimziations such as *general unboxing* and *late inlining*. These optimzations are sufficient to remove all allocations for a tail recursive variant our example program (1) (Boquist, 1999). However, for complex programs, implementing the Perceus reuse-analysis avoids additional allocations by updating nodes in-place.

Space leaks is another issue for lazy functional languages that can degrade performance and cause memory overflows (Peyton Jones, 1987). For example, our program (1) stack overflows for larger instances. We can prevent this by making sum tail recursive and eager in the first argument, by using the primForce primitive:

```
downFrom : ℕ → List ℕ
downFrom zero = []
downFrom (suc n) = n :: downFrom n

sum : ℕ → List ℕ → ℕ
sum m [] = m
sum m (n :: ns) =
  sum (primForce n _+_ m) ns

main = sum (downFrom 100)
```
(7)

Evaluating (primForce n _+_ m) prior to the recursive call, achieve several things. First, we prevent the stack overflow by not building and then evaluating a large suspended computation. Second, forcing _+_ drops the last reference count to n, and thus the object is deallocated. Which in turn, eliminates all out of memory errors and makes the worst-case heap consumption $\mathcal{O}(1)$. As this example demonstrates, selectively applying eager evaluation can drastically change the performance characteristics of our programs. However, this require a comprehensive understanding of the evaluation model and how it interacts with various compiler optimzations. Instead, we would like to develop a *strictness analysis* (Mycroft, 1980) that can automatically decide when to use eager evaluation. This is an intresting research opportunity because Agda is a total language so evaluation order does not matter. Therefore, the analyis can freely mix lazy and eager evaluation without accidentally making a productive program unproductive.

6 Related work

Reference counting implementations are uncommon in the literature of lazy functional languages. Goldberg (1988) uses a distributed reference counting scheme, simliar to that of Lermen and Maurer (1986), in the lazy functional language Alfalfa. Kaser et al. (1992) uses reference counting in a parallel lazy functional language called EQUALS. They observe that reference counting minimizes memory contention[5] because memory deallocations are distributed. This observervation is further supported by R. Jones and Lins (1996) and Peyton Jones (1987).

To our knowledge, precise reference counting which utilizes ownership to free objects as soon as possible have not yet been implemented in any lazy functional language. However, there have been multiple implementations for eager functional languages (Pinto, 2023; Reinking et al., 2021; Teeuwissen, 2023; Ullrich & de Moura, 2021). Teeuwissen's implementation in the Roc programming language is the most simliar to ours. Their algorithm is also based on Perceus and it is implemented in an intermediate representation with explicit memory operations. Moreover, they use an defunctionalization algorithm by Brandon et al. (2023) to compile higher-order functions. However, their algorithm specializes higher-order function for each callee. Compared to using branching, this gives better performance at the cost of larger code size.

The most recent implementation of GRIN is presented by Podlovics et al. (2021). Among other

[5] That is, multiple processes trying to access the same memory at once.

contributions, they implement an analyis by Turk (2010) that creates a bipartite graph of node producers and consumers. They use this analyis to perform a *dead data eliminatation* optimzation. Finally, Podlovics et al. (2021) presents overview of other attempts at using GRIN since Boquist's thesis.

7 Conclusions

In this paper, we combine lazy functional programming with precise reference counting. Our implementation compiles Agda to GRIN and extends GRIN with precise reference counting instructions. Then, we compile GRIN to LLVM and show that the program reclaims all the memory. Currently, our implementation allocates a lot of nodes. In future research, we would like to minimize allocations by implementing the rest of the GRIN optimzations and the Perceus reuse analysis. We are also interested in compiling a larger set of Agda programs, create a type system for GRIN, and develop a strictness analyis.

References

Augustsson, L. (1984). A compiler for lazy ml. *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 218–227. https://doi.org/10.1145/800055.802038

Boquist, U. (1995). Interprocedural register allocation for lazy functional languages. *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 270–281. https://doi.org/10.1145/224164.224215

Boquist, U. (1999). Code optimisation techniques for lazy functional languages.

Boquist, U., & Johnsson, T. (1996). The grin project: A highly optimising back end for lazy functional languages. *Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, 58–84.

Brandon, W., Driscoll, B., Dai, F., Berkow, W., & Milano, M. (2023). Better defunctionalization through lambda set specialization. *Proc. ACM Program. Lang.*, *7*(PLDI). https://doi.org/10.1145/3591260

Collins, G. E. (1960). A method for overlapping and erasure of lists. *Commun. ACM*, *3*(12), 655–657. https://doi.org/10.1145/367487.367501

Goldberg, B. (1988). Multiprocessor execution of functional programs. *International Journal of Parallel Programming*, *17*(5), 425–473. https://doi.org/10.1007/BF01383883

Hausmann, P. (2015). The agda uhc backend. https://api.semanticscholar.org/CorpusID:61450588

Johnsson, T. (1984). Efficient compilation of lazy evaluation. *SIGPLAN Not.*, *19*(6), 58–69. https://doi.org/10.1145/502949.502880

Johnsson, T. (1991). Analysing heap contents in a graph reduction intermediate language. In S. L. P. Jones, G. Hutton, & C. K. Holst (Eds.), *Functional programming, glasgow 1990* (pp. 146–171). Springer London.

Jones, P., L, S., & Peyton Jones, S. (1992). Implementing lazy functional languages on stock hardware: The spineless tagless g-machine (Journal of Functional Programming). *Journal of Functional Programming*, *2*, 127–202. https://www.microsoft.com/en-us/research/publication/implementing-lazy-functional-languages-on-stock-hardware-the-spineless-tagless-g-machine/

Jones, R., & Lins, R. (1996). *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc.

Kaser, O., Pawagi, S., Ramakrishnan, C. R., Ramakrishnan, I. V., & Sekar, R. C. (1992). Fast parallel implementation of lazy languages—the equals experience. *SIGPLAN Lisp Pointers*, *5*(1), 335–344. https://doi.org/10.1145/141478.141570

Lermen, C.-W., & Maurer, D. (1986). A protocol for distributed reference counting. *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 343–350. https://doi.org/10.1145/319838.319875

Mycroft, A. (1980). The theory and practice of transforming call-by-need into call-by-

value. *Proceedings of the Fourth 'Colloque International Sur La Programmation' on International Symposium on Programming*, 269–281.

Peyton Jones, S. L. (1987). *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc.

Pinto, E. (2023). Perceus for ocaml. https://www.eltonpinto.me/assets/work/mthesis-perceus-for-ocaml.pdf

Podlovics, P., Hruska, C., & Pénzes, A. (2021). A modern look at grin, an optimizing functional language back end. *Acta Cybernetica*, *25*(4), 847–876. https://doi.org/10.14232/actacyb.282969

Reinking, A., Xie, N., de Moura, L., & Leijen, D. (2021). Perceus: Garbage free reference counting with reuse. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. https://doi.org/10.1145/3453483.3454032

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. *Proceedings of the ACM Annual Conference - Volume 2*, 717–740. https://doi.org/10.1145/800194.805852

Teeuwissen, J. (2023). Reference counting with reuse in roc. https://studenttheses.uu.nl/handle/20.500.12932/44634

Turk, R. (2010). Perceus for ocaml. https://www.eltonpinto.me/assets/work/mthesis-perceus-for-ocaml.pdf

Turner, D. (1979). A new implementation technique for applicative languages. *Software: Practice and Experience*, *9*. https://api.semanticscholar.org/CorpusID:40541269

Ullrich, S., & de Moura, L. (2021). Counting immutable beans: Reference counting optimized for purely functional programming. *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. https://doi.org/10.1145/3412932.3412935

Wadler, P. (1984). Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 45–52. https://doi.org/10.1145/800055.802020

Wadler, P. (1988). Deforestation: Transforming programs to eliminate trees. *Proceedings of the Second European Symposium on Programming*, 231–248.

Wilson, P. R. (1992). Uniprocessor garbage collection techniques. *IWMM*. https://api.semanticscholar.org/CorpusID:206841815