

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 back-end for the Agda programming language with precise reference counting. The compiler uses Boquist and Johnsson’s intermediate language GRIN to compile lazy programs. GRIN uses 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 code generator.

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 computation¹ (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 their unintuitive control flow. In this paper, we adapt the Perceus algorithm to a new lazy compiler back-end for the Agda programming language and proof assistant. As a first step, we transform Agda into an intermediate language called GRIN (Johnsson, 1991).

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

¹Reference counted programs may introduce pauses similar to tracing garbage collectors. For example, when decrementing a long linked list all at once.

lexically scoped variables are stored in registers instead of on 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 (thunks) and higher-order functions are defunctionalized (Reynolds, 1972), by an interprocedural analysis called the heap points-to analysis. As a result, all function calls through opaque pointers are eliminated. This also means that all GRIN values are in weak head normal form. However, in order to distinguish between complex values (thunks and closures) and simple values (literals and data types), future mentions of *suspended computation* and *weak head normal form* refers to the representation in the Agda program.

2.1 Code generation

Currently, our compiler only accepts a subset of Agda which is lambda lifted, first-order, and monomorphic. Following is Agda program which computes the sum of the first 100 numbers.

```

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

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

main = sum (downFrom 100)

```

Our back-end starts by converting the program into an untyped lambda calculus with let and case expressions called Treeless (Hausmann, 2015). The Treeless language uses administrative normal form, which means that the case scrutinee is always a variable, and the patterns cannot be nested or overlap. Following is the Treeless representation of `downFrom`.

```

1 downFrom x7 = case x7 of
2   0 → []
3   _ → let x5 = 1

```

```

4       x4 = _-_ x7 x5
5       x3 = downFrom x4 in
6       _::_ x4 x3

```

The implementation uses de Bruijn indices to represent variables, but this paper uses variable names to make it more readable.

GRIN is similar to the Treeless syntax, but instead of let expressions we use the builtin state monad to bind variables and sequence operations. The monadic operations are **unit**, **store**, **fetch**, and **update**. The bind operator ";" is infix and right-associative. We can translate "**let** x = val **in** foo x" lazily by allocating the value and passing the pointer as an argument to the function: "**store** val ; λ x_{ptr} → foo x_{ptr}". Here, *val* must be a constant node value. A constant node is a tag followed by a sequence of arguments. 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`.

```

1 downFrom x7 =
2   eval x7 ; λ Cnat x6 →
3   case x6 of
4     0 → unit (C[])
5     _ →
6       store (Cnat 1) ; λ x5 →
7       store (F_-_ x7 x5) ; λ x4 →
8       store (FdownFrom x4) ; λ x3 →
9       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. The semantics values are a subset of syntactic values with an extra value representing undefined behaviour (\perp). Additionally, we have a new construct for heap allocated nodes. A heap allocated node consists of a reference count, a tag, and a sequence of values.

$term$	$::=$	$term ; \lambda lpat \rightarrow term$	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
$\{\dots\}$			means 0 or 1 times
$\{\dots\}^*$			means 0 or more times

Figure 1: *GRIN syntax.*

We have implemented the semantics as an interpreter, check that new transformations do not change the semantics. Due to the lower level of GRIN, we can collect information about resource usage. For instance, our example allocates the following nodes: 101 Cnat, 101 FdownFrom, 100 F_-, and 100 Fsum. 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.

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

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 suspended computations to its weak head normal form. There are no suspended computations in GRIN and all values are weak head normal form. Thus, "suspended computation" and "weak head normal form" should always refer to the corresponding representation in the source program. In the above function, "eval x7 ; λ Cnat x6 → ..." evaluates the value at the pointer (x7) to a boxed integer Cnat x6.

Eval inlining generates a specialized eval function for each call site. To evaluate a suspended computation, we load the node from the heap using **fetch**. Then, we pattern match on the possible nodes. Constructor nodes are already in weak head normal form so they are left unchanged. Function nodes need to be evaluated by applying the arguments to the corresponding function. Finally, the heap is updated with the evaluated value.

```

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 x34 ; λ () →
8     unit x35
9   ) ; λ Cnat x6 →
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)

```

Eval inlining require a set of possible nodes for each abstract heap location. The set needs to be relatively small, or otherwise an excessive amount of code will be generated. We use the *heap points-to analysis* (Johnsson, 1991). 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). Instead, this paper will only provide a general intuition of the algorithm. Consider the inlined evaluation in downFrom. There are two tags in the case expression: F_ _ and Cnat. F_ _ comes from the suspended recursive call inside downFrom, and Cnat is from the **update** operation and the suspended call to downFrom in the main function.

```

1 main =
2   store (Cnat 100) ; λ x20 →
3   store (FdownFrom x20) ; λ x19 →
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 code generator and are all implemented, except *inlining calls to apply* which is used for partially applied functions. The optimizing transformations significantly alter the program and achieve similar effects to deforestation (Wadler, 1988) and listlessness (Wadler, 1984). 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 the scope of this project. 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 . λ_1 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 brief a overview of the algorithm and discuss some challenges when adapting Perceus to GRIN. We also describe two optimizations: *drop specialization* and *dup/drop fusion*.

The algorithm uses two sets of resource tokens; an owned environment and a borrowed environment. Elements in the owned environment must be consumed exactly once. We consume a value by calling the function `drop`, or by transfer ownership to another consumer. An example of this is `main` which require no reference counting operations because it transfers ownership of both of its allocations. The allocation `"store (Cnat 100) ; λ x20 \rightarrow "` is consumed by the suspended computation `"store (FdownFrom x20) ; λ x19 \rightarrow "`, which in turn is consumed by `"sum x19"`. 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`. For example, we can conclude that the operations `fetch` and `update` must be borrowing operations. This is evident because both `fetch` and `update` use `x14` prior to it being explicitly dropped. We can also conclude that `fetch` bind variables that extend the owned environment. Meanwhile, the bound variables of function appli-

cations extend the borrowed environment.

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 regular values stored in registers. There is no point to reference count values in registers, so we need to treat pointer variables and non-pointer variables differently. This is in contrast to the calculus which Perceus is defined for, λ_1 , where all values except variables are heap-allocated (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 x59 x60 x61 \rightarrow ..."`. This is problematic because `x60` and `x61` are undefined when `downFrom` returns the empty list. Therefore, we are only allowed to `dup` the variables once the tag is known.

As a result of GRIN’s explicit memory operations, we can describe the Perceus primitives `dup`, `drop`, `is-unique`, `decreef`, and `free` with GRIN’s existing constructs. 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 possible tags to drop the child references. This is similar to the problem with a general `eval` function (Section 2.2). A crucial difference is that `drop` is recursive, so we cannot always eliminate the general function by specialization and inlining. In Figure 3b, we specialize both calls to `drop`. This eliminates all reference counting operations in the branch for the empty list. Then, we push down the `dup` operations and eliminate `dup/drop` pairs. This optimization is called *dup/drop fusion* and the result is presented in Figure 3c. We can 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 performs destructive updates when possible. This transformation is not yet implemented.

```

1 sum x14 =
2   fetch x14 [2] ; λ x40 →
3   downFrom x40 ; λ x59 x60 x61 →
4   case x59 of
5     [] →
6       update x14 (C[]) ; λ () →
7       drop x14 ; λ () →
8       unit (Cnat 0)
9   _::_ →
10    dup x61 ; λ () →
11    dup x60 ; λ () →
12    update x14 (C_::_ x60 x61) ; λ () →
13    drop x14 ; λ () →
14    store (Fsum x61) ; λ x10 →
15    _+_ x10 x61

```

(a) *dup/drop insertion*

<pre> 1 sum x14 = 2 fetch x14 [2] ; λ x40 → 3 downFrom x40 ; λ x59 x60 x61 → 4 case x59 of 5 [] → 6 update x14 (C[]) ; λ () → 7 fetch x14 [0] ; λ x499 → 8 (case x499 of 9 1 → free x14 10 - → 11 PSub x499 1 ; λ x498 → 12 update x14 [0] x498 13) ; λ () → 14 unit (Cnat 0) 15 _::_ → 16 dup x61 ; λ () → 17 dup x60 ; λ () → 18 update x14 (C_::_ x60 x61) ; λ () → 19 fetch x14 [0] ; λ x501 → 20 (case x501 of 21 1 → 22 drop x61 ; λ () → 23 drop x60 ; λ () → 24 free x14 25 - → 26 PSub x501 1 ; λ x500 → 27 update x14 [0] x500 28) ; λ () → 29 store (Fsum x61) ; λ x10 → 30 _+_ x10 x61 </pre>	<pre> sum x14 = fetch x14 [2] ; λ x40 → downFrom x40 ; λ 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) ; λ x10 → _+_ x10 x61 </pre>
--	--

(b) *drop specialization*

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

Figure 3: *Perceus transformations*

4 LLVM code generator

We have implemented a GRIN interpreter and an LLVM code generator to check 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 Result

```
{-# TERMINATING #-}
downFrom : ℕ → List ℕ
downFrom zero = []
downFrom (suc n) =
  primForce n (λ n → n :: downFrom n)

sum : ℕ → List ℕ → ℕ
sum acc [] = acc
sum acc (x :: xs) =
  sum (primForce x _+_ acc) xs

main = sum 0 (downFrom 100)
```

This is common space leak problem amongs lazy functional programs (Wadler, 1987)

6 Related work

Reference counting implementations are uncommon in the litterature of lazy functional languages. Kaser et al. (1992) uses a reference counting for

the lazy parallel language EQUALS. They observe that reference counting minimizes memory contention² because memory operations are interleaved throughtout the computation. Additionally, they note that their reference counting implementation achieved greatly reduced heap usage and good memory locality.

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. The Perceus algorithm have been implemented for multiple eager languages (Pinto, 2023; Reinking et al., 2021; Teeuwissen, 2023; Ullrich & de Moura, 2021).

7 Conclusions and future work

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 optimizing transformations and the Perceus reuse analysis. We are also interested in developing a type system for GRIN and compiling a larger set of Agda programs.

References

- 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.

²i.e. Multiple processes trying to access the same memory at once.

- 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.
- 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>
- 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, 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>
- Pinto, E. (2023). Perceus for ocaml. <https://www.eltonpinto.me/assets/work/mthesis-perceus-for-ocaml.pdf>
- 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>
- 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. (1987). Fixing some space leaks with a garbage collector. *Software: Practice and Experience*, 17(9), 595–608. <https://doi.org/https://doi.org/10.1002/spe.4380170904>
- Wadler, P. (1988). Deforestation: Transforming programs to eliminate trees. *Proceedings of the Second European Symposium on Programming*, 231–248.