Precise reference counting for lazy functional languages with interprocedural analysis

Martin Fredin

June 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 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 computation[1] (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 preciseness. Perceus is implemented in the functional language Koka, along with two 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 lexically scoped variables are stored in registers in-

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

stead 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 are shown in Figure 1. There are 4 kinds of constructs; terms, values, lambda patterns, and case patterns. All syntactically correct expressions are not valid. For example, the value at the function position of an application must be either a top-level function (*def*) or a primitive (*prim*). It cannot be a variable because there are no indirect function calls in GRIN. Likewise, top-level functions cannot be passed as arguments because GRIN is a first-order language.

2.1 Code generation

The current implementation of GRIN only compiles a subset of Agda which is lambda lifted, first order, and monomorphic.

```
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 the treeless syntax. The treeless syntax has explicit case expressions which use A-normal-form, meaning that the scrutinee is always a variable, and the alternatives cannot be nested or overlap. Following is the treeless representation of downFrom.

```
downFrom x1 = case x1 of
  0 → []
  _ → let x2 = 1
          x3 = _-_ x1 x2
          x4 = downFrom x3 in
      _::_ x3 x4
```

The implementation uses de Bruijn indices to represent variables, but this paper uses variable names to make it more readable. During this phase, we also transform the program so applications only take variables as operands.

GRIN is very similar to the treeless syntax, but instead of the 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 → foo x". 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.

```
downFrom x1 =
  eval x1 ; λ Cnat x2 →
  case x2 of
    0 → unit ([])
    _ →
      store (Cnat 1) ; λ x3 →
      store (F_-_ x1 x3) ; λ x4 →
      store (FdownFrom x4) ; λ x5 →
      unit (C_::_ x4 x5)
```

2.2 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 x1 ; λ Cnat x2 → ..." evaluates the value at the pointer (x1) to a boxed integer Cnat x2.

Eval inlining generates a specialized eval function for each call site. To evaluate a suspended

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.*

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.

```
downFrom x1 =
  (fetch x1 ; λ x6 →
   (case x6 of
      Cnat x7 → unit (Cnat x7)
      F_-_ x8 x9 → _-_ x8 x9
   ) ; λ x10 →
   update x1 x10 ; λ () →
   unit x10
  ) ; λ Cnat x2 →
  case x2 of
    0 → unit ([])
    _ →
      store (Cnat 1) ; λ x3 →
      store (F_-_ x1 x3) ; λ x4 →
      store (FdownFrom x4) ; λ x5 →
      unit (C_::_ x4 x5)
```

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 will 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.

```
main =
  store (Cnat 100) ; λ x11 →
  store (FdownFrom x11) ; λ x12 →
  sum x12 ; λ Cnat x13 →
  printf x13
```

Boquist's thesis contains 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. For the optimizing transformations, only *copy propagation* is implemented.

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 x_{11} →" is consumed by the suspended computation "**store** (FdownFrom x_{11}) ; λ x_{12} →", which in turn is consumed by "sum x_{12}". 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`.

To adapt Perceus to GRIN, TODO

1. Are all variables be reference counted?
2. Do bindings extend the borrowed environment or the owned environment?
3. Which operations consume references?

- One of the goals of GRIN is to improve register utilization for lazy functional lan-

guages (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 for variables are heap-allocated (Reinking et al., 2021). Another difference is that functions in GRIN are not allowed to return pointers. Moreover, all function calls return *explicit nodes* rather than node variables. For example: "`downFrom x40 ; λ x59 x60 x61 → ...`". This i problematic because `x60` and `x61` are undefined when `downFrom` returns the empty list.

- Another challenge is the builtin operations `fetch` and `update`, which uses a reference but does not consume it. This is similar to how dup does not consume their reference. However, one crucial difference is that dup do not interfere with algorithm. To solve this, we need to introduce borrowing for `fetch` and `update`.
- When a node is stored to the heap, the reference count is set to one and the allocating function is responsible for decrementing the counter and potentially freeing the object. A function may instead transfer this responsibility to another *consumer*.
- Ownership can be transferred to either another function or a node (suspended function call or a constructor). Consuming the resources A resource is consumed by calling the function `drop`.
- In GRIN, we have to be explicit about allocations, retrieving nodes from the heap, and overwriting heap nodes. As a result, the Perceus reference counting primitives `dup`, `drop`, `isunique`, `decref`, and `free` can be described with GRIN's exisiting constructs. [2] Although, GRIN does not have a primitive for freeing memory, this can be simulated by a function call to libc's `free`.
-
- See Figure 2.
- Crucial to specialize drops because of big drop function.

4 LLVM code generator

- GRIN in unopinonated about the node representation. There only on reguiremnt: there should be an easy way to extract the tag.
- Explain node structure `[4 x i64]`
- Short text about tailcalls
- Currenlty we do not need a type system because all functions return full nodes but after the general unboxing transformation this will change. Podlovics et al. (2021) have also developed a LLVM back-end for GRIN and they needed a type system for the mentioned reason.
- Currenlty we use libc's `malloc` and `free`, however, Pinto suggest that these should be implemented in assembly.
- Non-atomic reference count operations.

5 Result

To test that our compiler back-end actually works and that all memory is reclaimed, we implemented a GRIN interpreter and a LLVM code generator. We discovered that our back-end allocate many objects.

There are 402 allocations in our example program (page 2): 101 `Cnat` nodes, 101 `FdownFrom` nodes, 100 `F_-_` nodes, 100 `Fsum` nodes.

- Stack overflows (tail calls partly remedy this)
- Integer overflow
- The necessary parts of GRIN and Perceus is implemented but a lot of optimizations are left on the table. In GRIN we have mostly implemented the necessary simplifying transforma-

[2] This only possible in our slightly modified version of GRIN. Boquist's specification of GRIN could not overwrite individual fields of a heap node using `update`.

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

(a) *dup/drop insertion*

```
sum x14 =
  fetch x14 [2] ; λ x40 →
  downFrom x40 ; λ x59 x60 x61 →
  case 2 of
    [] →
      update x14 ([]) ; λ () →
      fetch 4 [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 (_::_ x60 x61) ; λ () →
      fetch 4 [0] ; λ x501 →
      (case x501 of
         1 →
           drop x61 ; λ () →
           drop x60 ; λ () →
           free 5
         _ →
           PSub x501 1 ; λ x500 →
           update 6 [0] x500
      ) ; λ () →
      store (Fsum x61) ; λ x10 →
      _+_ x10 x61
```

(b) *drop specialization*

```
sum x14 =
  fetch x14 [2] ; λ x40 →
  downFrom x40 ; λ x59 x60 x61 →
  case 2 of
    [] →
      update x14 ([]) ; λ () →
      fetch 4 [0] ; λ x499 →
      (case x499 of
         1 → free x14
         _ →
           PSub x499 1 ; λ x498 →
           update x14 [0] x498
      ) ; λ () →
      unit (Cnat 0)
    _::_ →
      update x14 (_::_ x60 x61) ; λ () →
      fetch 4 [0] ; λ x501 →
      (case x501 of
         1 →
           free 5
         _ →
           dup x61 ; λ () →
           dup x60 ; λ () →
           PSub x501 1 ; λ x500 →
           update 6 [0] x500
      ) ; λ () →
      store (Fsum x61) ; λ x10 →
      _+_ x10 x61
```

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

Figure 2: *Perceus...*

tions, which turns GRIN into a state which is suitable for the code generator. We haven't implemented drop specialization or heap reuse analysis.

6 Relevant Work

7 Conclusion and Future Work

- It would cool to benchmark our work but we lack many optimization.
- The current implementation of GRIN and Perceus have not yet implementation many optimization transformations. For example, GRIN lacks function inlining, generalized unboxing, and arity raising. Perceus lacks it two most import transformations; drop specialization and reuse analysis.
- Another huge optimization on is a strictness analysis.
- Add reuse and borrowing
- Utilse GRIN's whole program compilation strategy and the heap points-to analysis to statically determine unshared values during compile time, and thus minimizing the number of reference countinging operations.
- It would also be intresting to utilse 0-modality (erasure) in Agda's type system, and later also 1-modality when Agda gets it.
- In the current naive implementation drop enumerates all the possible tags

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.

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

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.

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 Cybern.*, *25*, 847–876.

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

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