# 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 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 a 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 a object, and reclaims the heap space once the reference count drops to zero. 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 ensure 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. To do this, we first 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 instead of the stack. Later, GRIN was reformulated

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

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 uses 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 treelss 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 it's weak head normal form. An example of this is "eval $x_1$ ; λ Cnat $x_2$ → ..." in the function above. In this example, the value at the pointer ($x_1$) is evaluated to a boxed integer Cnat $x_2$. 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 over 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 cor-

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

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

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

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

$\{...\}$ means 0 or 1 times

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

**Figure 1:** *GRIN syntax.*

responding 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 0
  ) ; 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_-_` comes from the (lazy) recursive call inside `downFrom`, and `Cnat` is from the **update** operation and the call to `downFrom` in the main function.

```
main =
  store (Cnat 100) ; λ x₁₁ →
  store (downFrom 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. 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. The reason for this is...

# 3 Precise reference counting

In GRIN, we have to be explicit about allocations, retrieving nodes from the heap, and overwriting heap nodes. As a result, the Perceus primitives `dup`, `drop`, `is-unique`, `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`.

# 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.

[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`.

- 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 transformations, 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