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). However, 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) makes reference counting more competitive by utilizing static guarantees to make the algorithm precise, which means that objects are deallocated as soon as possible. They present a formalized algorithm called Perceus which ensures that the reference counting is precise. 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

Johnsson (1991) 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 are shown in Figure 1. All syntactically correct expressions are not valid. For example, the value at the function position 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.

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

$term$	$::=$	$term\ ;\ \lambda lpat \to term$	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.*

2.1 Code generation

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

```
open import Agda.Builtin.Nat using (suc; zero; _+_) renaming (Nat to ℕ)
open import Data.List using (List; _::_; [])

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)
```

Figure 2: *Agda example program*

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. Below is the treeless representation of downFrom. Notice, that variable references use De Bruijn indicies which are represented by regular numbers, whereas the actual numbers are prefixed with "#". We also transform the program so applications only takes variables as operands.

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

GRIN is very similar to the treelss syntax, so the translation is straigthforward.

```
downFrom x₁ =
eval 0 ; λ Cnat x₂ →
case 0 of
  0 → unit ([])
  _ →
    store (Cnat #1) ; λ x₃ →
    store (F_-_ 2 0) ; λ x₄ →
    store (FdownFrom 0) ; λ x₅ →
    unit (C_::_ 1 0)
```

2.2 Analysis and transformations

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. Also, the *introduce registers* transformations is implemented but unused as it conflicts with LLVM. For the optimizing transformations, only *copy propagation* is implemented.

3 Precise reference counting

In GRIN, we have to be explicit about allocations, retriving 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

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.

[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`.

There are 402 allocations in our example program (Figure 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 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.

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