Optimizing lazy functional languages with precise reference counting

Martin Fredin

June 2023

1 Introduction

All values in purely functional languages are immutable. Immutability is important because it limits shared state, and makes it easier to reason about the program. In practice, this means that a function such as map returns a new list instead of modifying the input.

$$
\begin{aligned}
&\mathsf{map} : (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B \\
&\mathsf{map}\ f\ [] = [] \\
&\mathsf{map}\ f\ (x :: xs) = f\ x :: \mathsf{map}\ f\ xs
\end{aligned}
$$

This is great if the input list (List A) is used later in the program. Otherwise, it is better to reuse the allocation for List A by updating the list in place. This avoids both the deallocation of List A and the allocation of List B. To preserve the semantics of the program, destructive updates should only be perfomed on values which are garanteed to be non-shared. We call a value shared if there is multiple references to it.

Reference counting (**collins1960**) 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 value, and reclaims the heap space once the reference count drops to zero. As a result of the interleaved collection strategy, memory usage remain low and the throughput is continuous throughout the computation[1] (**jones1996**). However, ...

[1] Not entirely true.

- Lower throughput than tracing GC
- Not very good at handling short lived values which is most values (especially in pure FP)
- **reinking2021<empty citation>** and **ullrich2021<empty citation>** minimize RC operations through precise reference counting and a linear calculus.
- more...

2 Graph Reduction Intermediate Notation

$term, t$	::=	$term$; $\lambda \rightarrow lalt$	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

Figure 1: GRIN syntax.

All syntactically correct expressions are not valid. For example, the value at the function position must be either a top-level function or a primitive. 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 langauge.

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 Result

5 Relevant Work

6 Conclusion and Future Work

- 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.
- Currently decref and dup are primitives is GRIN however it would be intresting to adjust update with offsets, similiar to fetch.
- In the current naive implementation drop enumerates all the possible tags

[2] This only possible in our slightly modified version of GRIN. **boquist1999**'s original specification of GRIN could not overwrite individual fields of a heap node using `update`.