Garbage Collection for Reversible Functional Languages

Torben Ægidius Mogensen(✉)

DIKU, University of Copenhagen, Universitetsparken 5,
DK-2100 Copenhagen O, Denmark
torbenm@diku.dk

Abstract. Reversible languages are programming languages where all programs can run both forwards and backwards. Reversible functional languages have been proposed that use symmetric pattern matching and data construction. To be reversible, these languages require linearity: Every variable must be used exactly once, so no references are copied and all references are followed exactly once. Copying of values must use deep copying. Similarly, equality testing requires deep comparison of trees.

A previous paper describes reversible treatment of reference counts, which allows sharing of structures without deep copying, but there are limitations. Applying a constructor to arguments creates a new node with reference count 1, so pattern matching is by symmetry restricted to nodes with reference count 1. A variant pattern that does not change the reference count of the root node is introduced to allow manipulation of shared data. Having two distinct patterns for shared and unshared data, however, adds a burden on the programmer.

We observe that we can allow pattern matching on nodes with arbitrary reference count if we also allow constructor application to return nodes with arbitrary reference counts. We do this by using maximal sharing: If a newly constructed node is identical to an already existing node, we return a pointer to the existing node (increasing its reference count) instead of allocating a new node with reference count 1.

To avoid searching the entire heap for an identical node, we use hash-consing to restrict the search to a small segment of the heap. We estimate how large this segment needs to be to give a very low probability of allocation failure when the heap is less than half full. Experimentally, we find that overlapping segments gives dramatically better results than disjoint segments.

1 Introduction

A reversible first-order functional language RFUN [16] has been suggested. Steps towards an implementation were made by first implementing a simple heap manager [1] and later [6], a full translation of RFUN to reversible machine language was made. This translation uses linearity and deep copying, so all heap nodes have exactly one reference. Other reversible functional languages that rely on linearity include Theseus [7] and Ψ-Lisp [2].

J. Krivine and J.-B. Stefani (Eds.): RC 2015, LNCS 9138, pp. 79–94, 2015.
DOI: 10.1007/978-3-319-20860-2_5

By extending Axelsen's heap manager [1], a previous paper [12] studied reversible treatment of reference counts, which allows copying of values by pointer sharing. This did, however, not constitute true garbage collection, as the functional language used in this paper required explicit distinction of shared and unshared nodes, so unshared nodes are explicitly deallocated at pattern matching and shared nodes are explicitly preserved at pattern matching, using two different forms of pattern for constructor nodes with one or several references.

In this paper, we will implement a heap manager that does true garbage collection, so high-level languages using this manager do not have to distinguish between shared and unshared nodes and nodes are automatically collected once the last reference is used. In order to make construction and deconstruction symmetric we must, however, make construction use maximal sharing: If a node identical to the node being built exists anywhere in the heap, no new node is built. Instead, a new pointer to the existing node is returned and its reference count is increased. Conversely, deconstructing (by pattern matching) a shared node removes a pointer to the node and deconstructing an unshared node deallocates it. To make searching for existing identical nodes efficient, we will employ hash-consing [4,5], which limits the search space to a small segment of the heap. We estimate how large this segment must be to make the probability of allocation failure very small if the heap is less than half full. We find that a segment that can contain four to eight nodes suffice for a 32-bit address space.

We implement heap operations including a node construction/deconstruction procedure in a low-level reversible intermediate language RIL.

2 The Reversible Intermediate Language RIL

We define a reversible low-level language RIL, similar to a language of the same name in [13]. RIL is inspired by Janus [10], using unstructured jumps in the style of the Janus variant described in [11]. We use RIL instead of Janus because it is closer to a machine language but not specific to any particular machine architecture. RIL can be considered as a reversible alternative to three-address code and is mainly a vehicle for presenting code in a machine-independent form. Its design is in itself not a significant contribution.

A RIL program consists of an unordered set of basic blocks, each consisting of an entry point followed by either updates and exchanges or a subroutine call and is terminated by an exit point. We will describe each of these below.

RIL uses 32-bit words using two's complement number representation. Addresses are to 8-bit bytes, but will be truncated to the nearest 32-bit boundary at memory transfers. We will use an unbounded number of named variables to represent registers, relying on register allocation to map these to a finite set of numbered registers.

2.1 Entry and Exit Points

An entry point has one of the forms

$l \leftarrow$	where l is a label,
$l_1; l_2 \leftarrow c$	where c is a condition and l_1 and l_2 are labels, or
`begin` l	where l is a label.

An exit point has one of the forms:

$\rightarrow l$	where l is a label,
$c \rightarrow l_1; l_2$	where c is a condition and l_1 and l_2 are labels, or
`end` l	where l is a label.

Each label in the program must occur in exactly one entry point and exactly one exit point. Furthermore, a label that occurs in a `begin` entry point must also occur in an `end` exit point.

Conditions are of the form $L \bowtie R$, where a left-value L is either a named variable x or of the form $M[x]$, representing the memory location pointed to by a variable x. A right-value R is either a left-value or a signed constant in the range -2^{31} to $2^{31} - 1$, and $\bowtie$ is an operator from the set `==`, `<`, `>`, `!=`, `<=`, `>=` and `&`, using notation from the programming language C. We use 0 to represent false and any non-zero value to represent true, so the condition L & R is true if the result of the bitwise AND is non-zero.

`begin` and `end` represent beginnings and ends of subroutines. The start and end of the entire program are entry and exit points with the label `main`. An exit point of the form $\rightarrow l$ constitutes an unconditional jump to the (unique) entry point where l occurs. An exit point of the form $c \rightarrow l_1; l_2$ constitutes a conditional jump: If c is true, the jump goes to l_1, otherwise to l_2. An entry point of the form $l \leftarrow$ unconditionally accepts incoming jumps. An entry point of the form $l_1; l_2 \leftarrow c$ conditionally accepts incoming jumps: Jumps to l_1 are accepted if c is true and jumps to l_2 are accepted if c is false. If the incoming jump is not accepted, a run-time error occurs.

2.2 Updates and Exchanges

A basic block can hold a (possibly empty) sequence of updates and exchanges.

An update is of the form $L \oplus= \ R_1 \ \odot \ R_2$, where L is a left-value, R_1 and R_2 are right-values and $\oplus=$ is one of the update assignments `+=`, `-=` or `^=` with the same semantics as in the programming language C. $\odot$ is an infix arithmetic operation that can be either `+`, `-`, `^`, `&`, `|`, `>>`, or `<<`, again with the same semantics as in C. Specifically, `&` and `|` are bitwise AND/OR and `>>` and `<<` are bitwise shifts.

An exchange is of the form $L_1 \leftrightarrow L_2$, where L_1 and L_2 are left-values. The effect is that the values in the two specified locations are swapped.

In order to ensure reversibility, the following restrictions apply to updates and exchanges:

- In an update of the form $L \oplus= \ R_1 \ \odot \ R_2$, the same named variable can not occur both to the left and to the right of the update operator $\oplus=$.

- In an update of the form $L \oplus\!= R_1 \odot R_2$, memory accesses (left-values of the form $M[x]$) can not be used on both sides of the update operator $\oplus\!=$.
- In an exchange of the form $L_1 \leftrightarrow L_2$, the same named variable can not occur both to the left and to the right of the exchange operator $\leftrightarrow$.

2.3 Subroutine Calls

Instead of exchanges and updates, a basic block can hold a single subroutine call. A subroutine call is done using the instructions `call` l and `uncall` l. There can be several calls to the same subroutine. We use an implicit stack to store return information.

A subroutine call must be in a basic block of the form $l_1 \leftarrow$ `call` $l \rightarrow l_2$ or $l_1 \leftarrow$ `uncall` $l \rightarrow l_2$.

In such a block, `call` l stores l_2 on the implicit stack and jumps to the entry point `begin` l until it reaches `end` l, at which point it pops the stack and jumps to the label l_2 that is stored on the top of the stack.

RIL (like Janus) also supports running subroutines backwards: `uncall` l stores l_2 on the implicit stack, and then runs the subroutine l backwards, starting from the exit point `end` l and ending with `begin` l, again returning via the stack to l_2.

2.4 Formal Semantics

Figure 1 shows a formal semantics for execution of RIL as rules for state transitions. A state consists of the program P (which never changes), an environment ρ, that maps named variables to integers, a memory store σ, that maps word-aligned addresses to integers, a stack S that stores return labels, and the current label l. A transition of the form $P\ \rho\ \sigma\ S\ l \rightleftharpoons P\ \rho'\ \sigma'\ S\ l'$ states that a state $P\ \rho\ \sigma\ S\ l$ will lead to the state $P\ \rho'\ \sigma'\ S\ l'$ in one or more steps.

The semantics shows how execution of a basic block makes a transition from a label to another while changing the environment and store. The transition is bidirectional, so it describes both forwards and backwards execution. This is used in the rule for `uncall`, where the transition relation is used in the reverse order for executing the subroutine.

We use I to indicate an unspecified instruction (update or exchange), E to indicate an unspecified entry point, X to indicate an unspecified exit point and c to indicate an unspecified condition. Abusing notation, we use $\oplus$, $\odot$ and $\bowtie$ to represent both the syntactic and semantic versions of operators. We use the same rules for evaluating expressions and conditions, using 0 to represent false and any non-zero value to represent true.

2.5 Shorthands

To make code more readable, we introduce a number of shorthands when displaying RIL code in the paper.

We will use $L \oplus\!= R$ as an abbreviation of $L \oplus\!= R$ `+` 0.

Basic blocks:

$$\frac{(l_1 \leftarrow\ \texttt{call}\ l\ \rightarrow l_2) \in P \quad P\ \rho\ \sigma\ (l_2 : S)\ l \rightleftharpoons P\ \rho'\ \sigma'\ (l_2 : S)\ l}{P\ \rho\ \sigma\ S\ l_1 \rightleftharpoons P\ \rho'\ \sigma'\ S\ l_2}$$

$$\frac{(l_1 \leftarrow\ \texttt{uncall}\ l\ \rightarrow l_2) \in P \quad P\ \rho'\ \sigma'\ (l_2 : S)\ l \rightleftharpoons P\ \rho\ \sigma\ (l_2 : S)\ l}{P\ \rho\ \sigma\ S\ l_1 \rightleftharpoons P\ \rho'\ \sigma'\ S\ l_2}$$

$$\frac{(E\ I\ X) \in P \quad E \vdash \rho\ \sigma\ l_1 \quad I \models \rho\ \sigma \leftrightharpoons \rho'\ \sigma' \quad X \dashv \rho'\ \sigma'\ l_2}{P\ \rho\ \sigma\ S\ l_1 \rightleftharpoons P\ \rho'\ \sigma'\ S\ l_2}$$

$$\frac{P\ \rho\ \sigma\ S\ l_1 \rightleftharpoons P\ \rho'\ \sigma'\ S\ l_2 \quad P\ \rho'\ \sigma'\ S\ l_2 \rightleftharpoons P\ \rho''\ \sigma''\ S\ l_3}{P\ \rho\ \sigma\ S\ l_1 \rightleftharpoons P\ \rho''\ \sigma''\ S\ l_3}$$

Entry points:

$$\frac{}{l\ \leftarrow\ \ \vdash \rho\ \sigma\ l} \qquad \frac{}{\texttt{begin}\ l\ \vdash \rho\ \sigma\ l} \qquad \frac{\rho\ \sigma \rhd c \rightsquigarrow 0}{l_1; l_2\ \leftarrow\ c \vdash \rho\ \sigma\ l_2} \qquad \frac{\rho\ \sigma \rhd c \not\rightsquigarrow 0}{l_1; l_2\ \leftarrow\ c \vdash \rho\ \sigma\ l_1}$$

Exit points:

$$\frac{}{\rightarrow\ l\ \dashv \rho\ \sigma\ l} \qquad \frac{}{\texttt{end}\ l\ \dashv \rho\ \sigma\ l} \qquad \frac{\rho\ \sigma \rhd c \rightsquigarrow 0}{c\ \rightarrow\ l_1; l_2\ \dashv \rho\ \sigma\ l_2} \qquad \frac{\rho\ \sigma \rhd c \not\rightsquigarrow 0}{c\ \rightarrow\ l_1; l_2\ \dashv \rho\ \sigma\ l_1}$$

Updates and exchanges:

$$\frac{\rho\ \sigma \rhd e \rightsquigarrow v \quad w = u \oplus v}{x \oplus= e \rhd \rho[x \mapsto u]\ \sigma \leftrightharpoons \rho[x \mapsto w]\ \sigma}$$

$$\frac{\rho\ \sigma \rhd e \rightsquigarrow v \quad w = u \oplus v}{M[x] \oplus= e \rhd \rho[x \mapsto a]\ \sigma[a \mapsto u] \leftrightharpoons \rho[x \mapsto a]\ \sigma[a \mapsto w]}$$

$$\frac{}{x \leftrightarrow y \rhd \rho[x \mapsto u,\ y \mapsto v]\ \sigma \leftrightharpoons \rho[x \mapsto v,\ y \mapsto u]\ \sigma}$$

$$\frac{}{x \leftrightarrow M[y] \rhd \rho[x \mapsto u,\ y \mapsto a]\ \sigma[a \mapsto v] \leftrightharpoons \rho[x \mapsto v,\ y \mapsto a]\ \sigma[a \mapsto u]}$$

$$\frac{}{\models \rho\ \sigma \leftrightharpoons \rho\ \sigma} \qquad \frac{I_1 \models \rho\ \sigma \leftrightharpoons \rho'\ \sigma' \quad I_2 \models \rho'\ \sigma' \leftrightharpoons \rho''\ \sigma''}{I_1\, I_2 \models \rho\ \sigma \leftrightharpoons \rho''\ \sigma''}$$

Expressions and conditions:

$$\frac{k \text{ is a constant}}{\rho\ \sigma \rhd k \rightsquigarrow k} \qquad \frac{}{\rho[x \mapsto v]\ \sigma \rhd x \rightsquigarrow v} \qquad \frac{}{\rho[x \mapsto a]\ \sigma[a \mapsto v] \rhd M[x] \rightsquigarrow v}$$

$$\frac{\rho\ \sigma \rhd R_1 \rightsquigarrow v_1 \quad \rho\ \sigma \rhd R_2 \rightsquigarrow v_2 \quad w = v_1 \odot v_2}{\rho\ \sigma \rhd R_1 \odot R_2 \rightsquigarrow w}$$

$$\frac{\rho\ \sigma \rhd R_1 \rightsquigarrow v_1 \quad \rho\ \sigma \rhd R_2 \rightsquigarrow v_2 \quad w = v_1 \bowtie v_2}{\rho\ \sigma \rhd R_1 \bowtie R_2 \rightsquigarrow w}$$

Fig. 1. Semantics of RIL

Two blocks $E\ I_1\ \rightarrow l$ and $l \leftarrow l\ I_2\ X$, where E in an entry point, X is an exit point and I_1 and I_2 are updates, exchanges or calls, will be abbreviated to a single extended basic block $E\ I_1\, I_2\ X$. This abbreviation can be applied repeatedly, so arbitrarily many basic blocks connected by unconditional exit and entry point pairs will be shown as a single extended block. Similarly, when a block with a conditional exit point where the second label (corresponding to a false condition) occurs in an unconditional entry point of another block, these will be merged and the conditional exit point will be shown as a conditional jump with one target (and fall-through when the condition is false). For example, a block ending with $c \rightarrow l_1; l_2$ will be merged to a block starting with $l_2 \leftarrow$ joined by the one-way jump $c \rightarrow l_1$. Symmetrically, a block with an unconditional exit point can be merged with another block with a conditional entry point: $\rightarrow l_2$ and $l_1; l_2 \leftarrow c$ are joined to $l_1 \leftarrow c$.

Additionally, we will at entry and exit points for subroutines, i.e., after `begin` and before `end`, add assertions of the form `assert` c, where c is a condition. Variables occurring in these assertions are the input and output parameters for the subroutine, and they specify preconditions and postconditions for the subroutine and can be seen either as comments that specify an invariant or as conditions that are actively checked at runtime. In the latter case, an assertion `assert` c can be expanded into a conditional jump and a conditional entry point: $c \rightarrow l_1; l2$ and $l_1; l_2 \leftarrow$ `true`, where `true` represents any tautology, e.g., x`==`x. We will in assertions, additionally, allow conjunction of simple conditions using the `&&` operator. Such conjunctions can be expanded to sequences of simple assertions.

3 Implementation of a Heap Manager

We will now show implementations of a heap manager in RIL, using the shorthands described in Section 2.5. In particular, we use assertions to describe partial pre and post conditions[1]. If code compiled from a high-level language to RIL statically ensures that these assertions are true, they can be omitted.

3.1 Data Representation

Our heap manager will use LISP-like values. A value can be either a symbol, an integer or a pair of two values a and d, written as `Cons(`a`,`d`)`. We represent these in the following way on a machine with 32-bit words:

- The value 0 is used for uninitialised variables and heap nodes.
- An integer is represented by a machine word ending in 1. The integer value is given by the first 31 bits.
- A symbol is represented by a machine word that ends in 10.
- `Cons(`a`,`d`)` is represented by a three-word-aligned address A (so ending in 00), where $H \leq a \leq lastH$, where $lastH$ is the address of the last node in the heap. A points to a node consisting of three words: A reference count

[1] The assertions are not strong enough to describe full pre and post conditions.

and two fields a and d, representing the elements of a pair `Cons(`a`,`d`)`. In an unallocated node, all three words are zero, and in an allocated node, all three words are non-zero.

We assume that the fields a and d in a pair `Cons(`a`,`d`)` can not be destructively updated, which is true in functional languages. This allows us to share identical pairs.

Note that we can test if a value v is a node pointer by checking if the two least significant bits are 00, i.e., that the condition v `& 3` is 0, representing false.

3.2 Value Copying

This subroutine copies a value stored in the variable `copyP` to the variable `copyQ`, which must initially be zero, while maintaining reference counts if the value is a pointer.

```
begin copy
assert copyP > 0 && copyQ == 0
copyP & 3 → copyNonPointer
M[copyP] += 1
copyNonPointer ← copyP & 3
copyQ += copyP
assert copyP > 0 && copyQ == copyP
end copy
```

Note that only copying of pointer fields update reference counts.

Calling `copy` in reverse requires that the two values are identical. Though the reference count of the node decreases when calling `copy` in reverse, it can never reach zero, as the equality assertion implies that there are at least two pointers to the node before the decrement.

3.3 Copying the Fields of a `Cons` Node

We sometimes want to access the fields of a `Cons` node while keeping the pointer to the node. This will not change the reference count to the node, but it will increase the reference count to its fields (if they are pointers).

```
begin fields
assert fieldsP >= H && fieldsA == 0 && fieldsD == 0
fieldsP += 4
fieldsA += M[fieldsP]
fieldsA & 3→ nonPointerA
M[fieldsA] += 1
nonPointerA ← fieldsA & 3
fieldsP += 4
fieldsD += M[fieldsP]
fieldsD & 3 → nonPointerD
M[fieldsD] +=1
nonPointerD ← fieldsD & 3
fieldsP -= 8
assert fieldsP >= H && fieldsA > 0 && fieldsD > 0
end fields
```

When called in reverse, it is implicitly assumed that `fieldsA` and `fieldsD` are equal to the fields of `fieldsP`. `fieldsA` and `fieldsD` are cleared by subtracting from these the two fields of `fieldsP` and reducing the reference counts of these (if pointers). If, when `fields` is called in reverse, `fieldsA` and `fieldsD` are *not* equal to the fields of `fieldsP`, the assertion that they are zero after `begin fields` will fail.

3.4 Naive Implementation of Construction / Destruction of Nodes

The above subroutines neither allocate nor free data, as reference counts are non-zero both before and after calling these subroutines.

We now describe a subroutine `cons` that takes two arguments `consA` and `consD` and returns a pointer `consP` to a `Cons`-node that has the values of `consA` and `consD` as fields, while clearing the contents of these variables.

If there is already such a node in the heap, a new reference to this node is returned. Finding an existing node with the required fields requires a search through the heap. If there is no suitable node to share, a new node is allocated. Allocating a new node requires searching backwards through the heap for a node that has zero reference count (which also implies zeroed fields).

When called in reverse, `cons` takes a pointer `consP` and returns the values of the fields in the variables `consA` and `consD`, while clearing `consP`. If the node pointed to by `consP` is unshared (indicated by reference count 1), it is deallocated by clearing the reference count and the fields to 0. The code for `cons` is shown in Figure 2. For readability, we will use indentation to indicate structure.

The loop `consSearchSame` searches forwards through the heap to find a matching node. If that succeeds, the block `consFoundSame` increases the reference count of the node and decreases the counts of the fields (if they are not symbols) because it clears `consA` and `consD`, that are references to the fields.

If the search for a matching node fails, the loop `consSearchEmpty` searches for an unallocated node. If one such is found, a new `Cons`-node is created in it.

If no empty node is found, no allocation is possible, and a jump to the label `consFail` is made. This should do some kind of error handling (not shown).

It should be obvious that the naive implementation of `cons` is slow: Whenever a new node is created, the entire heap is walked through to find an existing, identical node, and if that fails, the heap is walked through again to find an unallocated node. Since allocations happen near the top of the heap and searches for existing nodes start from the bottom, the average case is quite bad. So we will, below, describe an optimised implementation of `cons`.

3.5 An Optimised Implementation of `Cons`

We will use an old idea for effective maximal sharing: Hashing field values to find the address of the node [5]. Rather than searching the entire heap, we search only a small segment of the heap starting at the address given by a hash code calculated from the values of the fields. Since we both add and remove nodes,

```
begin cons
assert consA != 0 && consD != 0 && consP == 0
consP += H
consSearchSame ← consP > H
  M[consP] == 0 → consNext
    consP += 4
    M[consP] != consA → consNotA
      consP += 4
      M[consP] == consD → consFoundSame
      consP -= 4
    consNotA ← M[consP] != consA
    consP -= 4
  consNext ← M[consP] == 0
  consP += 12
consP <= lastH → consSearchSame
consSearchEmpty ← consP <= lastH
  consP -= 12
  consP < H → consFail
M[consP] != 0 → consSearchEmpty
  M[consP] += 1
  consP += 4
  consA ↔ M[consP]
  consP += 4
  consD ↔ M[consP]
  consP -= 8
consEnd ← M[consP] > 1
assert consP >= H && consA == 0 && consD == 0
end cons

consFoundSame ←
  consD & 3 → consNonPointerD
    M[consD] -= 1
  consNonPointerD ← consD & 3
  consD -= M[consP]
  consP -= 4
  consA & 3 → consNonPointerA
    M[consA] -= 1
  consNonPointerA ← consA & 3
  consA -= M[consP]
  consP -= 4
  M[consP] += 1
→ consEnd
```

Fig. 2. Naive reversible implementation of `cons`

we can not keep searching until we find either a match or a free node, so we need a fixed segment size. For efficiency reasons, we want the segment to be small, but we also want to minimise the risk that the segment we want to allocate into is full while the heap as a whole is mostly empty. It seems reasonable to require that the probability of trying to allocate into an already full segment is very small when less than half the total heap is allocated.

If hashing distributes values uniformly randomly, allocating m Cons-nodes in a heap with n segments of size b is equivalent to randomly throwing m balls into n bins, where no bin holds more than b balls. If $n < m < n \log n$, the maximum number of balls in a bin is [15] with very high probability no more than $\frac{\log n}{\log \frac{n \log n}{m}}$. If the heap is half full, $m = \frac{bn}{2}$, and we get a bound of $\frac{\log n}{\log \frac{2 \log n}{b}}$. We want this not to exceed b, so we want $\frac{\log n}{\log \frac{2 \log n}{b}} \leq b \Leftrightarrow \frac{n}{\log n} \leq \frac{e^b}{b}$. For $b = 12$, we get $n \leq 1.6 \cdot 10^5$, which gives approximately $12 \cdot 1.6 \cdot 10^5 = 1.8 \cdot 10^6$ nodes.

We will assume a subroutine `hash` exists that takes a cleared variable `hashV` and the values in `consA` and `consD` as arguments and returns in the variable `hashV` a hash code of `consA` and `consD` while preserving the values of `consA` and `consD`. If `hashV` is the hash code for `consA` and `consD`, running `hash` in reverse will clear `hashV`. The hash code should be the start of a segment, *i.e.*, $H, H+12b, H+24b, \ldots,$ `lastH`$-(12b-12)$ (the address of the last segment in the heap). We will in Section 4 discuss how the `hash` procedure can be implemented.

The optimised implementation of `cons` is shown in Figure 3. We start by computing `hashV`, which is the address of the start of the segment to search, and `segEnd`, which is the address of the last node in this segment. We then search as before, but constrained to the interval between `hashV` and `segEnd`. When we find the matching or empty node we need, we uncompute `hashV` and `segEnd`.

4 Reversible Hashing

We want a procedure `hash` that expects variables `consA` and `consD` to contain non-zero values and `hashV` to be zero. After the call, `consA` and `consD` are unchanged and `hashV` holds a value between H and $lastH - (12b - 12)$ in increments of $12b$. We will assume that b is a power of 2, as this eases scaling.

We base our hash function, shown in Figure 4, on Jenkins' 96-bit reversible mix function [8] that is well tested and has good statistical properties. This mixes three integers, so we use a constant as the third. The three values are stored in variables `hashA`, `hashB` and `hashC` that are globally initialised to constants k_a, k_b and k_c. `hashA` and `hashB` are XOR'ed with `consA` and `consD`, Jenkins' mix function is executed, and a scaled version of the resulting `hashC` is used as the hash code `hashV`. Uncalling `hash` resets `hashA`, `hashB` and `hashC` to their original values and `hashV` to zero.

Since `hashV` needs to be between H and $lastH - (12b - 12)$ in increments of $12b$ and `hashC` can be any 32-bit integer, we need to mask and scale this to the right range. We first choose H and $lastH$ so $H = lastH - 12b \cdot 2^m + 12$ for some m. We can then do the scaling by bitwise ANDing `hashC` with $b \cdot 2^{m+2} - 4b$,

```
begin cons
assert consA != 0 && consD != 0 && consP == 0 && hashV == 0 && segEnd == 0
call hash
consP += hashV
segEnd += hashV + $(12b - 12)$
consSearchSame ← consP > hashV
  M[consP] == 0 → consNext
    consP += 4
    M[consP] != consA → consNotA
      consP += 4
      M[consP] == consD → consFoundSame
      consP -= 4
    consNotA ← M[consP] != consA
    consP -= 4
  consNext ← M[consP] == 0
  consP += 12
consP <= segEnd → consSearchSame
consSearchEmpty ← consP <= segEnd
  consP -= 12
  consP < $H$ → consFail
M[consP] != 0 → consSearchEmpty
  segEnd -= hashV + $(12b - 12)$
  uncall hash
  M[consP] += 1
  consP += 4
  consA ↔ M[consP]
  consP += 4
  consD ↔ M[consP]
  consP -= 8
consEnd ← M[consP] > 1
assert consP >= $H$ && consA == 0 && consD == 0 && hashV == 0 && segEnd == 0
end cons

consFoundSame ←
  segEnd -= hashV + $(12b - 12)$
  uncall hash
  consD & 3 → consNonPointerD
    M[consD] -= 1
  consNonPointerD ← consD & 3
  consD -= M[consP]
  consP -= 4
  consA & 3 → consNonPointerA
    M[consA] -= 1
  consNonPointerA ← consA & 3
  consA -= M[consP]
  consP -= 4
  M[consP] += 1
→ consEnd
```

Fig. 3. Optimised reversible implementation of `cons`

giving $4b$ times an m-bit integer. We then multiply this by 3 (getting a multiple of $12b$) and add H. The maximum value will, hence, be $3(b \cdot 2^{m+2} - 4b) + H = 12b \cdot 2^m - 12b + lastH - 12b \cdot 2^m + 12 = lastH - (12b - 12)$, as we wanted.

```
begin hash
assert hashV == 0 && hashA == $k_a$ && hashB == $k_b$ && hashC == $k_c$
hashA ^= consA
hashB ^= consD
hashA -= hashB + hashC
hashA ^= hashC >> 13
hashB -= hashC + hashA
hashB ^= hashA << 8
hashC -= hashA + hashB
hashC ^= hashB >> 12
hashA -= hashB + hashC
hashA ^= hashC >> 12
hashB -= hashC + hashA
hashB ^= hashA << 16
hashC -= hashA + hashB
hashC ^= hashB >> 5
hashA -= hashB + hashC
hashA ^= hashC >> 3
hashB -= hashC + hashA
hashB ^= hashA << 10
hashC -= hashA + hashB
hashC ^= hashB >> 15
hashV += hashC & $(b \cdot 2^{m+2} - 4)$
hashV += hashC & $(b \cdot 2^{m+2} - 4)$
hashV += hashC & $(b \cdot 2^{m+2} - 4)$
hashV += $H$
end hash
```

Fig. 4. Reversible `hash` subroutine based on Jenkins' mix function

5 Performance Analysis and Experiments

When analysing the time used by `cons`, we count the number of instructions executed. We do not count `assert` instructions, as these are assumed to be invariants that need not be checked at runtime, and we will count an unconditional jump to an unconditional entry point as free (as code layout can in most cases make it so), but we count a conditional jump to a conditional entry point as two instructions because two conditions are checked. We also count `call` and `uncall` as two instructions each, as we count the cost of the return into the cost of the call, but otherwise we do not distinguish the cost of instructions. This is, admittedly, a gross simplification, but a more precise costs measure depends on the choice of machine that RIL is translated to and the cost model of this.

```
begin hash
assert hashV == 0 && hashT == $k_a$
hashT ^= consA << 7
hashT += consA >> 1
hashT ^= consD << 5
hashT += consD >> 3
hashV += hashT & $(b \cdot 2^{m+2} - 4)$
hashV += hashT & $(b \cdot 2^{m+2} - 4)$
hashV += hashT & $(b \cdot 2^{m+2} - 4)$
hashV += $H$
end hash
```

Fig. 5. Simplified `hash` subroutine

In the best case, a call to `cons` will have symbols as arguments and find a match in the first node it encounters. This will use 71 instructions, 52 of which are used by the two calls to `hash`. If the arguments to `cons` are pointers, add two instructions to update their reference counts. In the worst case, no matching node is found (but `consA` matches the head of all nodes in the segment) and the only free node is the last searched. This will use $15b + 58$ instructions, 52 of which are, again, used by `hash`.

We have tested the heap manager with different heap sizes to find how full the heap is when an allocation fails. This is (rather naively) done by adding pseudo-random numbers to a list until allocation fails. For each heap size, we have run the test twenty times using different random numbers (so hashing yields different numbers), and for each heap size we have listed the average and maximum number of free nodes when allocation fails. Our first test uses $b = 8$.

Heap size (nodes)	average free nodes		maximum free nodes		spread	
2^{10}	135	13%	639	62%	177	17%
2^{14}	3139	19%	10291	63%	2891	18%
2^{18}	56373	22%	144783	55%	40769	16%
2^{22}	1014423	24%	2194523	52%	713904	17%

The worst-case utilisation is with all heap sizes under 50%, though the average case is around 80%. Our estimate was that a bin size of 12 would be needed for heaps up to 10^6 nodes, so it is hardly surprising that the results are bad. The reason we have chosen a smaller bin size is to compare the setup above with a variant where the bins overlap: Instead of using hash values that point to the start of disjoint 8-node segments, we change it so hash values can point to any node start in the heap. We still use a bin size of 8, so the changes to the code are minimal: The mask used in the `hash` subroutine needs changing, so it instead of $b \cdot 2^{m+2} - 4b$ is $b \cdot 2^{m+2} - 4$. Additionally, $b - 1$ extra nodes must be added to the heap, so there are b nodes to search from the largest generated address onwards. The results are shown below

Heap size (nodes)	average free nodes		maximum free nodes		spread	
2^{10}	69	7%	171	17%	52	5%
2^{14}	767	5%	2170	13%	553	3%
2^{18}	13708	5%	34229	13%	12403	5%
2^{22}	384618	9%	974626	23%	239922	6%

The difference is quite dramatic: In none of the tests was more than 23% of the heap unused, and the average heap utilisation is between 91% and 95%, depending on heap size.

Hashing takes a significant fraction of the time for allocating nodes, so using a simpler hashing function might be worthwhile, even if this gives a higher risk of collision. As an experiment, we have replaced the hash function with the very simple function shown in Figure 5 and repeated the above tests. Reducing the body of `hash` from 24 to 8 instructions reduces the number of instructions for executing `cons` by 32 (as `hash` is called twice). The results of using the simplified `hash` function are shown below.

Heap size (nodes)	average free nodes		maximum free nodes		spread	
2^{10}	79	8%	215	21%	73	7%
2^{14}	438	3%	1022	6%	372	2%
2^{18}	10266	4%	24673	9%	7484	3%
2^{22}	215346	5%	597394	14%	168784	4%

The results are not significantly different from the previous, but that may be due to the simplicity of the tests. More testing is required to verify if this or another cheap hash function is adequate for more realistic use.

We have also made experiments using a smaller segment size, i.e., searching only 4 nodes instead of 8 for matching or empty nodes. This will reduce the cost of executing `cons`, but the expectation is that this will make allocation failure happen when a larger fraction of the heap is empty. The table below shows heap utilisation with a segment size of 4 (using the simple hash function).

Heap size (nodes)	average free nodes		maximum free nodes		spread	
2^{10}	107	10%	293	29%	94	9%
2^{14}	1311	8%	6546	40%	1860	11%
2^{18}	15871	6%	43784	17%	12874	5%
2^{22}	802967	19%	1884954	45%	539114	13%

The heap utilisation is, as expected, not as good as with the larger segment size, but it is not below 50% in any of the tests. So if space is not tight, it can be a good choice to reduce the segment size to 4.

6 Conclusion and Discussion

We have presented a reversible intermediate language RIL and implementations in RIL of a reversible heap manager that uses reference counts and hash-consing to achieve garbage collection: The heap manager does all the necessary management of reference counts, and nodes are automatically reclaimed when their reference count becomes zero.

The key insight is that to get symmetric construction and destruction of values, either linearity (no sharing) or maximal sharing is needed. Previous works assume linearity, but we implement maximal sharing by a reversible hash-cons subroutine. This allows copying of values just by copying pointers (and updating reference counts) and structural equality testing by comparison of pointers. This is, we believe, the first real reversible garbage collection method that does not rely on linearity.

Our use of fixed-size segments (bins) to handle hash-code collisions means that a segment can be filled long before the heap is full. We have calculated a segment size that makes this very unlikely before the heap is at least half full. The calculation is based on results from the literature for non-overlapping bins and gives a fairly large segment size. We have also tested overlapping bins, which has not been studied much in the literature, and found that the results are dramatically better when overlapping bins are used. Overlapping bins have proved to be beneficial for cuckoo hashing [9], so it is, perhaps, not so surprising. Experimentally, we have found that a segment size of 8 gives heap utilisation above 75% in the worst case and better than 90% in the average case, while a segment size of 4 gives heap utilisation around 55% in the worst case and better than 80% in the average case.

The hashing and searching used during node construction has a significant cost, so construction and deconstruction of nodes is relatively expensive. This is, however, partially offset by vey cheap equality testing and copying of data.

We have made preliminary tests of two different hash functions and found no significant difference in results, though one is much simpler than the other. Further trials with more realistic allocation/freeing patterns are needed to draw a firm conclusion. Further trials could also investigate more different hash functions.

Our use of a fixed segment/bin size makes reversibility simple at the cost of relatively low heap utilisation. More advanced hashing techniques such as two-way chaining [3] may improve heap utilisation with small segment sizes at the cost of increasing the hashing cost. Cuckoo hashing [9,14] offer high utilisation with short searches, but this (or any other) hash-table technique that moves nodes around after they are allocated are not suitable for our purpose, as changing the address of a node requires modifying pointers globally.

A limitation of the heap manager is that heap nodes can only be pairs. It is easy enough to modify the heap manager to another fixed size of nodes, but mixing nodes of several sizes in the same heap will require all nodes to be padded to the largest size. A simple solution is to have separate heaps for different node sizes, but that can be very wasteful. Larger tuples can be built from pairs, but that requires an average of one node per field in the tuple. A compromise might be to let nodes be four words including reference count. This will waste one word when building pairs but there will be less waste when building larger tuples, as an average of two fields can be stored in each node. A node size that is a power of two can also save some instructions when scaling the hash code to a multiple of the node size.

References

1. Axelsen, H.B., Glück, R.: Reversible representation and manipulation of constructor terms in the heap. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 96–109. Springer, Heidelberg (2013)
2. Baker, H.G.: Nreversal of fortune — the thermodynamics of garbage collection. In: Bekkers, Y., Cohen, J. (eds.) Memory Management. Lecture Notes in Computer Science, vol. 637, pp. 507–524. Springer, Berlin Heidelberg (1992)
3. Broder, A.Z., Mitzenmacher, M.: Using multiple hash functions to improve IP lookups. In: Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001), vol. 3, pp. 1454–1463. IEEE Comput. Soc. Press (2001)
4. Ershov, A.P.: On programming of arithmetic operations. Communications of the ACM **1**(8), 3–6 (1958)
5. Goto, E.: Monocopy and associative algorithms in an extended lisp. Technical Report TR 74–03, University of Tokyo (1974)
6. Hansen, J.S.K.: Translation of a reversible functional programming language. Master's thesis, DIKU, University of Copenhagen, December 2014
7. James, R.P., Sabry, A.: Theseus: a high-level language for reversible computation. In: Reversible Computation - Booklet of work-in-progress and short reports (2014). http://www.reversible-computation.org
8. Jenkins, B.: Hash functions. Dr. Dobb's Journal of Software Tools **22**(7) (1997)
9. Lehman, E., Panigrahy, R.: 3.5-Way cuckoo hashing for the price of 2-and-a-bit. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 671–681. Springer, Heidelberg (2009)
10. Lutz, C.: Janus: a time-reversible language. A letter to Landauer (1986). http://www.tetsuo.jp/ref/janus.pdf
11. Mogensen, T.Æ.: Partial evaluation of janus part 2: assertions and procedures. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 289–301. Springer, Heidelberg (2012)
12. Mogensen, T.Æ.: Reference counting for reversible languages. In: Yamashita, S., Minato, S. (eds.) RC 2014. LNCS, vol. 8507, pp. 82–94. Springer, Heidelberg (2014)
13. Oh, C.W.: Reversible intermediate language for the translation of reversibleprogramming languages. Master's thesis, DIKU, University of Copenhagen, November 2009
14. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
15. Raab, M., Steger, A.: "Balls into bins" - a simple and tight analysis. In: Rolim, J.D.P., Serna, M., Luby, M. (eds.) RANDOM 1998. LNCS, vol. 1518, pp. 159–170. Springer, Heidelberg (1998)
16. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 14–29. Springer, Heidelberg (2012)