Perceus: Garbage Free Reference Counting with Reuse


Alex Reinking
Microsoft Research, USA
alex_reinking@berkeley.edu

Ningning Xie
University of Hong Kong
nnxie@cs.hku.hk

Leonardo de Moura
Microsoft Research, USA
leonardo@microsoft.com

Daan Leijen
Microsoft Research, USA
daan@microsoft.com

Abstract
We introduce Perceus, an algorithm for precise reference counting with reuse and specialization. Starting from a functional core language with explicit control-flow, Perceus emits precise reference counting instructions such that programs are garbage free, where only live references are retained. This enables further optimizations, like reuse analysis that allows for guaranteed in-place updates at runtime. This in turn enables a novel programming paradigm that we call functional but in-place (FBIP). Much like tail-call optimization enables writing loops with regular function calls, reuse analysis enables writing in-place mutating algorithms in a purely functional way. We give a novel formalization of reference counting in a linear resource calculus, and prove that Perceus is sound and garbage free. We show evidence that Perceus, as implemented in Koka, has good performance and is competitive with other state-of-the-art memory collectors.

1 Introduction
Reference counting [5], with its low memory overhead and ease of implementation, used to be a popular technique for automatic memory management. However, the field has broadly moved in favor of generational tracing collectors [28], partly due to various limitations of reference counting, including cycle collection, multi-threaded operations, and expensive in-place updates.

In this work we take a fresh look at reference counting. We consider a programming language design that gives strong compile-time guarantees in order to enable efficient reference counting at runtime. In particular, we build on the pioneering reference counting work in the Lean theorem prover [42], but we view it through the lens of language design, rather than purely as an implementation technique.

We demonstrate our approach in the Koka language [20, 22]: a functional language with mostly immutable data types together with a strong type and effect system. In contrast to the dependently typed Lean language, Koka is general-purpose, with support for exceptions, side effects, and mutable references via general algebraic effects and handlers [36, 37]. Using recent work on evidence translation [46, 47], all these control effects are compiled into an internal core language with explicit control flow. Starting from this functional core, we can statically transform the code to enable efficient reference counting at runtime. In particular:

- Due to explicit control flow, the compiler can emit precise reference counting instructions where a reference is dropped as soon as possible; we call this garbage free reference counting as only live data is retained (Section 2.2).
- We show that precise reference counting enables many optimizations, in particular drop specialization which removes many reference count operations in the fast path (Section 2.3), reuse analysis which updates (immutable) data in-place when possible (Section 2.4), and reuse specialization which removes many in-place field updates (Section 2.5). The reuse analysis shows the benefit of a holistic approach: even though the surface language has immutable data types with strong guarantees, we can use dynamic run-time information, e.g. whether a reference is unique, to update in-place when possible.
- The in-place update optimization is guaranteed, which leads to a new programming paradigm that we call FBIP: functional but in-place (Section 2.6). Just like tail-call optimization lets us write loops with regular function calls, reuse analysis lets us write in-place mutating algorithms in a purely functional way. We showcase this approach by implementing a functional version of in-order Morris tree traversal [32], which is stack-less, using in-place tree node mutation via FBIP.
- We present a formalization of general reference counting using a novel linear resource calculus, $\lambda^1$, which is closely based on linear logic (Section 3), and we prove that reference counting is sound for any program in the linear resource calculus. We then present the Perceus$^1$ algorithm as a deterministic syntax-directed version of $\lambda^1$, and prove

---

$^1$Perceus, pronounced per-see-us, is a loose acronym of "PrEcise Reference Counting with rEUse and Specialization".

\*The first two authors contributed equally to this work
that it is both sound (i.e. never drops a live reference), and
garbage free (i.e. only retains reachable references).
• We demonstrate Perceus by providing a full implementa-
tion for the strongly typed functional language Koka [1].
The implementation supports typed algebraic effect han-
dlers using evidence translation [47] and compiles into
standard C11 code. The use of reference counting means
no runtime system is needed and Koka programs can read-
ily link with other C/C++ libraries.
• We show evidence that Perceus, as implemented for Koka,
competes with other state-of-the-art memory collectors
(Section 4). We compare our implementation in allocation
intensive benchmarks against OCaml, Haskell, Swift,
and Java, and for some benchmarks to C++ as well. Even
though the current Koka compiler does not have many
optimizations (besides the ones for reference counting), it
has outstanding performance compared to these mature
systems. As a highlight, on the tree insertion benchmark,
the purely functional Koka implementation is within 10%
of the performance of the in-place mutating algorithm in
C++ (using std::map [10]).

Even though we focus on Koka in this paper, we believe that
Perceus, and the FBIP programming paradigm we identify,
are both broadly applicable to other programming languages
with similar static guarantees for explicit control flow.

2 Overview
Compared to a generational tracing collector, reference count-
ing has low memory overhead and is straightforward to
implement. However, while the cost of tracing collectors
is linear in the live data, the cost of reference counting is
linear in the number of reference counting operations. Opti-

mizing the total cost of reference counting operations is
therefore our main priority. There are at least three known
problems that make reference counting operations expensive
in practice and generally inferior to tracing collectors:
• Concurrency: when multiple threads share a data structure,
reference count operations need to be atomic, which is
expensive.
• Precision: common reference counted systems are not pre-
cise and hold on to objects too long. This increases memory
usage and prevents aggressive optimization of many re-
ference count operations.
• Cycles: if object references form a cycle, the runtime needs
to handle them separately, which re-introduces many of
the drawbacks of a tracing collector.

We handle each of these issues in the context of an eager,
fusional language using immutable data types together
with a strong type and effect system. For concurrency, we
precisely track when objects can become thread-shared (Sec-
tion 2.7.2). For precision, we introduce Perceus, our algorithm
for inserting precise reference counting operations that can
be aggressively optimized. In particular, we eliminate and
fuse many reference count operations with drop specializa-
tion (Section 2.3), turn functional matching into in-place
updates with reuse analysis (Section 2.4), and minimize field
updates with reuse specialization (Section 2.5).

Finally, although we currently do not supply a cycle collec-
tor, our design has two important mitigations. First, (co)in-
ductive data types and eager evaluation prevent cycles outside
of explicit mutable references, and it is statically known where
cycles can possibly be introduced in the code (Section 2.7.4).
Second, being a mostly functional language, mutable refer-
ences are not often used, and on top of that, reuse analysis
greatly reduces the need for them since in-place mutation is
typically inferred.

The reference count optimizations are our main contribu-
tion and we start with a detailed overview in the following
sections, ending with details about how we mitigate the
impact of concurrency and cycles.

2.1 Types and Effects
We start with a brief introduction to Koka [20, 22] – a strongly
typed, functional language that tracks all (side) effects. For
example, we can define a squaring function as:

fun square( x : int ) : total int { x * x }

Here we see two types in the result: the effect type total and
the result type int. The total type signifies that the func-
tion can be modeled semantically as a mathematically total
function, which always terminates without raising an excep-
tion (or having any other observable side effect). Effectful
functions get more interesting effect types, like:

define( x : int, y : int ) : exn int

where println has a console effect and divide may raise an
exception (exn) when dividing by zero. It is beyond the scope
of this paper to go into full detail, but a novel feature of Koka
is that it supports typed algebraic effect handlers which can
define new effects like async/await, iterators, or co-routines
without needing to extend the language itself [21–23].

Koka uses algebraic data types extensively. For example,
we can define a polymorphic list of elements of type a as:

type list(a) {
        Cons( head : a, tail : list(a) )
        Nil
    }

We can match on a list to define a polymorphic map function
that applies a function f to each element of a list xs:

fun map( xs : list(a), f : a -> e b ) : e list(b) {
    match(xs) {
        Cons(x,xx) -> Cons(f(x), map(xx,f))
        Nil  -> Nil
    }
}

Here we transform the list of generic elements of type a to
a list of generic elements of type b. Since map itself has no
intrinsic effect, the overall effect of map is polymorphic, and
equals the effect e of the function f as it is applied to every
element. The map function demonstrates many interesting
aspects of reference counting and we use it as a running example in the following sections.

2.2 Precise Reference Counting
An important attribute that sets Perceus apart is that it is precise: an object is freed as soon as no more references remain. By contrast, common reference counting implementations tie the liveness of a reference to its lexical scope, which might retain memory longer than needed. Consider:

```plaintext
fun foo() {
  val ys = map(xs, inc) // increment elements
  print(ys)
}
```

Many compilers emit code similar to:

```plaintext
fun foo() {
  val ys = list(1,1000000) // create large list
  val ys = map(xs, inc) // increment elements
  print(ys)
}
```

where we use a gray background for generated operations. The `drop(xs)` operation decrements the reference count of an object and, if it drops to zero, recursively drops all children of the object and frees its memory. These "scoped lifetime" reference counts are used by the C++ `shared_ptr` (calling the destructor at the end of the scope), Rust’s `Rc` (using the `Drop` trait), and Nim (using a `finally` block to call `destroy`) [48]. It is not required by the semantics, but Swift typically emits code like this as well [11].

Implementing reference counting this way is straightforward and integrates well with exception handling where the drop operations are performed as part of stack unwinding. But from a performance perspective, the technique is not always optimal: in the previous example, the large list `xs` is retained in memory while a new list `ys` is built. Both exist for the duration of `print`, after which a long, cascading chain of drop operations happens for each element in each list.

Perceus takes a more aggressive approach where ownership of references is passed down into each function: now `map` is in charge of freeing `xs`, and `ys` is freed by `print`: no `drop` operations are emitted inside `foo` as all local variables are consumed by other functions, while the `map` and `print` functions drop the list elements as they go. In this example, Perceus generates the code for `map` as given in Figure 1b. In the `Cons` branch, first the head and tail of the list are dupped, where a `dup(x)` operation increments the reference count of an object and returns itself. The `drop(xs)` then frees the initial list node. We need to `dup f` as well as it is used twice, while `x` and `xx` are consumed by `f` and `map` respectively.

At first blush, this seems more expensive than the scoped approach but, as we will see, this change enables many further optimizations. More importantly, transferring ownership, rather than retaining it, means we can free an object immediately when no more references remain. This both increases cache locality and decreases memory usage. For `map`, the memory usage is halved: the list `xs` is deallocated while the new list `ys` is being allocated.

2.3 Drop Specialization
Once we change to precise, ownership-based reference counting, there are many further optimization opportunities. After the initial insertion of `dup` and `drop` operations, we perform a drop specialization pass. The basic `drop` operation is defined in pseudocode as:

```plaintext
fun drop(x) {
    if (is-unique(x)) then drop children of x; free(x)
    else decref(x)
}
```

and drop specialization essentially inlines the `drop` operation specialized at a specific constructor. Figure 1c shows the drop specialization of our `map` example. Note that we only apply drop specialization if the children are used, so no specialization takes place in the `Nil` branch.

Again, it appears we made things worse with extra operations in each branch, but we can perform another transformation where we push down `dup` operations into branches followed by standard `dup/drop` fusion where corresponding `dup/drop` pairs are removed. Figure 1d shows the code that is generated for our `map` example.

After this transformation, almost all reference count operations in the fast path are gone. In our example, every node in the list `xs` that we map over is unique (with a reference count of 1) and so the `if` `(is-unique(xs))` test always succeeds, thus immediately freeing the node without any further reference counting.

2.4 Reuse Analysis
There is more we can do. Instead of freeing `xs` and immediately allocating a fresh `Cons` node, we can try to reuse `xs` directly as first described by Ullrich and de Moura [42]. Reuse analysis is performed before emitting the initial reference counting operations. It analyses each `match` branch, and tries to pair each matched pattern to allocated constructors of the same size in the branch. In our example, `xs` is paired with the `Cons` constructor. When such pairs are found, and the matched object is not live, we generate a drop-reuse operation that returns a `reuse token` that we attach to any constructor paired with it:

```plaintext
fun map(xs, f) {
    match(xs) {
      Cons(x,xx) {
        val ru = drop-reuse(xs)
        Cons@ru( f(x), map(xx, f))
      }
      Nil -> Nil
    }
}
```

The `Cons@ru` annotation means that (at runtime) if `ru==NULL` then the ` Cons` node is allocated fresh, and otherwise the memory at `ru` is of the right size and can be used directly. Figure 1e shows the generated code after reference count insertion.
fun map(xs : list a, f : a → e b) : e list b {
  match(xs) {
    Cons(x, xx) → Cons(f(x), map(xx, f))
    Nil → Nil
  }
}

(a) A polymorphic map function

fun map(xs, f) {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx)
      if (is-unique(xs))
        then drop(x); drop(xx); free(xs)
      else decref(xs)
      Cons(dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f); Nil
    }
  }
}

(b) dup/drop insertion (2.2)

c) drop specialization (2.3)

fun map(xs, f) {
  match(xs) {
    Cons(x, xx) {
      dup(x); dup(xx);
      val ru = if (is-unique(xs))
        then &xs
      else decref(xs);
      Cons(ru(dup(f)(x), map(xx, f)))
    }
    Nil {
      drop(xs); drop(f); Nil
    }
  }
}

(e) reuse token insertion (2.4)

fun map(xs, f) {
  match(xs) {
    Cons(x, xx) {
      val ru = if (is-unique(xs))
        then &xs
      else decref(xs);
      CONS@ru(dup(f)(x), map(xx, f))
    }
    Nil {
      drop(xs); drop(f); Nil
    }
  }
}

(f) drop-reuse specialization (2.4)

g) push down dup and fusion (2.4)

Fig. 1. Drop specialization and reuse analysis for map.

Compared to the program in Figure 1b, the generated code now consumes xs using drop-reuse(xs) instead of drop(xs).

Just like with drop specialization we can also specialize drop-reuse. The drop-reuse operation is specified in pseudocode as:

fun drop-reuse(x) {
  if (is-unique(x)) then drop children of x; &x
  else decref(x); NULL
}

where &x returns the address of x. Figure 1f shows the code for map after specializing the drop-reuse. Again, we can push down and fuse the dup operations, which finally results in the code shown in Figure 1g. In the fast path, where xs is uniquely owned, there are no more reference counting operations at all! Furthermore, the memory of xs is directly reused to provide the memory for the Cons node for the returned list – effectively updating the list in-place.

2.5 Reuse Specialization

The final transformation we apply is reuse specialization, by which we can further reuse unchanged fields of a constructor. A constructor expression like CONS@ru(x, xx) is implemented in pseudocode as:

fun CONS@ru(x, xx) {
  if (ru!=NULL)
    then { ru->head := x; ru->tail := xx; ru } // in-place
  else Cons(x, xx) // malloc’d
}

However, for our map example there would be no benefit to specializing as all fields are assigned. Thus, we only specialize constructors if at least one of the fields stays the same. As an example, we consider insertion into a red-black tree [14]. We define red-black trees as:

type color { Red; Black }

type tree {
  Leaf
  Node(color: color, left: tree, key: int, value: bool, right: tree)
}

The red-black tree has the invariant that the number of black nodes from the root to any of the leaves is the same, and that a red node is never a parent of red node. Together this ensures that the trees are always balanced. When inserting nodes, the invariants need to be maintained by rebalancing the nodes when needed. Okasaki’s algorithm [34] implements this elegantly and functionally (the full algorithm can be found in Appendix A):

fun bal-left(l : tree, k : int, v : bool, r : tree ) : tree {
  match(l) {
    Node(_, Node(Red, l, x, kx, vx, rx), ky, vy, r)
    → Node(Red, Node(Black, l, x, kx, vx, rx), ky, vy, Node(Black, r, k, v, r))
    ...
  }
}

fun ins(t : tree, k : int, v : bool) : tree {
  match(t) {
    Leaf → Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r) // second branch
Fig. 2. Morris in-order tree traversal algorithm in C.

```c
void inorder(tree root, void (*f)(tree t)) {
    tree* cursor = root;
    while (cursor != NULL) {
        if (cursor->left != NULL) {
            // no left tree, go down the right
            f(cursor->value);
            cursor = cursor->right;
        } else {
            // has a left tree
            tree* pre = cursor->left; // find the predecessor
            while (pre->right != NULL && pre->right != cursor) {
                pre = pre->right;
            }
            if (pre->right == NULL) {
                // first visit, remember to visit right tree
                pre->right = NULL;
                f(cursor->value);
                pre = pre->right;
            } else {
                // already set, restore
                f(cursor->value);
                pre->right = NULL;
                cursor = cursor->right;
            }
        }
    }
}
```

For this kind of program, reuse specialization is effective. For example, if we look at the second branch in ins we see that the newly allocated Node has almost all of the same fields as t except for the left tree l which becomes ins(l,k,v). After reuse specialization, this branch becomes:

```c
Node(Black, l, kx, vx, r) -> if (k < kx && is-unique(t)) then &
    then bal-left(ins(l,k,v), kx, vx, r)
...
```

In the fast path, where t is uniquely owned, t is reused directly, and only its left child is re-assigned as all other fields stay unchanged. This applies to many branches in this example and saves many assignments.

Moreover, the compiler inlines the bal-left function. At that point, every matched Node constructor has a corresponding Node allocation – if we consider all branches we can see that we either match one Node and allocate one, or we match three nodes deep and allocate three. With reuse analysis this means that every Node is reused in the fast path without doing any allocations!

Essentially this means that for a unique tree, the purely functional algorithm above adapts at runtime to an in-place mutating re-balancing algorithm (without any further allocation). Moreover, if we use the tree persistently [33], and the tree is shared or has shared parts, the algorithm adapts to copying exactly the shared spine of the tree (and no more), while still rebalancing in place for any unshared parts.

Fig. 3. FBIP in-order tree traversal algorithm in Koka.

2.6 A New Paradigm: Functional but In-Place (FBIP)

The previous red-black tree rebalancing showed that with Perceus we can write algorithms that dynamically adapt to use in-place mutation when possible (and use copying when used persistently). Importantly, a programmer can rely on this optimization happening, e.g. they can see the match patterns and match them to constructors in each branch.

This style of programming leads to a new paradigm that we call FBIP: “functional but in place”. Just like tail-call optimization lets us describe loops in terms of regular function calls, reuse analysis lets us describe in-place mutating imperative algorithms in a purely functional way (and get persistence as well). Consider mapping a function f over all elements in a binary tree in-order:

```koka
type visitor {
    Done
    BinR(right:tree, value : int, visit : visitor)
    BinL(left:tree, value : int, visit : visitor)
}
type direction { Up; Down }

fun tmap( f : int -> int, t : tree, visit : visitor, d : direction ) : tree {
    match(d) {
        Down -> match(t) { // going down a left spine
            Bin(1,x,r) -> tmap(f,1,Bin(r,x,visit),Down) // A
            Tip -> tmap(f,Tip,visit,Up) // B
        }
        Up -> match(visit) { // go up through the visitor
            Done -> t // C
            BinL(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
            BinL(1,x,t) -> tmap(f,Bin(1,x,t),v,Up) // E
        }
    }
}
```

This is already quite efficient as all the Bin and Tip nodes are reused in-place when t is unique. However, the tmap function is not tail-recursive and thus uses as much stack space as the depth of the tree.

In 1968, Knuth posed the problem of visiting a tree in-order while using no extra stack- or heap space [19] (for readers not familiar with the problem it might be fun to try this in your favorite imperative language first and see that it is not easy to do). Since then, numerous solutions have appeared in the literature. A particularly elegant solution was proposed by Morris [32]. This is an in-place mutating algorithm that swaps pointers in the tree to “remember” which parts are unvisited. It is beyond this paper to give a full explanation, but a C implementation is shown in Figure 2. The traversal essentially uses a right-threaded tree to keep track of which nodes to visit. The algorithm is subtle, though.
Since it transforms the tree into an intermediate graph, we need to state invariants over the so-called Morris loops [26] to prove its correctness.

We can derive a functional and more intuitive solution using the FBIP technique. We start by defining an explicit visitor datatype structure that keeps track of which parts of the tree we still need to visit. In Koka we define this data type as visitor given in Figure 3. (Interestingly, our visitor data type can be generically derived as a list of the derivative of the tree data type\(^2\) [17, 27]). We also keep track of which direction we are going, either Up or Down the tree.

We start our traversal by going downward into the tree with an empty visitor, expressed as \(\text{map}(f, \text{t, Done, Down})\). The key idea is that we are either Done (C), or, on going downward in a left spine we remember all the right trees we still need to visit in a BinR \((A)\) or, going upward again \((B)\), we remember the left tree that we just constructed as a BinL while visiting right trees \((D)\). When we come back \((E)\), we restore the original tree with the result values. Note that we apply the function \(f\) to the saved value in branch \(B\) (as we visit in-order), but the functional implementation makes it easy to specify a pre-order traversal by applying \(f\) in branch \(A\), or a post-order traversal by applying \(f\) in branch \(E\).

Looking at each branch we can see that each Bin matches up with a BinR, each BinR with a BinL, and finally each BinL with a Bin. Since they all have the same size, if the tree is unique, each branch updates the tree nodes in-place at runtime without any allocation, where the visitor structure is effectively overlaid over the tree nodes while traversing the tree. Since all \(\text{map}\) calls are tail calls, this also compiles to a loop and thus needs no extra stack- or heap space.

Finally, just like with re-balancing tree insertion, the algorithm as specified is still purely functional: it uses in-place updating when a unique tree is passed, but it also adapts gracefully to the persistent case where the input tree is shared, or where parts of the input tree are shared, making a single copy of those parts of the tree.

### 2.7 Static Guarantees and Language Features

So far we have shown that precise reference counting enables powerful analyses and optimizations of the reference counting operations. In this section, we use Koka as an example to discuss how strong static guarantees at compile-time can further allow the precise reference counting approach to be integrated with non-trivial language features.

#### 2.7.1 Non-Linear Control Flow

An essential requirement of our approach is that programs have explicit control flow so that it is possible to statically determine where to insert \(\text{dup}\) and \(\text{drop}\) operations. However, it is in tension with functions that have non-linear control flow, e.g. may throw an exception, use a \text{longjmp}, or create an asynchronous continuation that is never resumed. For example, if we look at the code for \text{map} before applying optimizations, we have:

```haskell
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx); drop(xs); dup(f)
      Cons( f(x), map(xx, f) )
    }
  }
}
```

If \(f\) raised an exception and directly exited the scope of \text{map}, then \(xx\) and \(f\) would leak and never be dropped. This is one reason why a C++ \text{shared_ptr} is tied to lexical scope; it integrates nicely with the stack unwinding mechanism for exceptions that guarantees each \text{shared_ptr} is dropped eventually.

In Koka, we guarantee that all control-flow is compiled to explicit control-flow, so our reference count analysis does not have to take non-linear control-flow into account. This is achieved through \text{effect typing} (Section 2.1) where every function has an effect type that signifies if it can throw exceptions or not. Functions that can throw are compiled into functions that return with an explicit error type that is either \text{Ok}, or \text{Error} if an exception is thrown. This is checked and propagated at every invocation\(^3\).

For example, for \text{map} the compiled code (before optimization) becomes like:

```haskell
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx); drop(xs); dup(f)
      match(f(x)) {
        Error(err) -> ( drop(xx); Error(err) )
        Ok(y) -> ( match(map(xx, f) )
          Error(err) -> drop(y); Error(err)
        Ok(y) -> Cons(y,ys)
      }
    }
  }
}
```

At this point all errors are explicitly propagated and all control-flow is explicit again. Note the we have no reference count operations on the \text{error} values as these are implemented as \text{value} types which are not heap allocated. This is similar to error handling in Swift \[18\] (although it requires the programmer to insert a \text{try} at every invocation), and also similar to various C++ proposals \[40\] where exceptions become explicit error values.

The example here is specialized for exceptions but the actual Koka implementation uses a generalized version of this technique to implement a multi-prompt delimited control monad \[15\] instead, which is used in combination with evidence translation \[47\] to express general algebraic effect handlers (which in turn subsume all other control effects, like exceptions, async/await, probabilistic programming, etc).

\(^2\)Conor McBride [27] describes how we can generically derive a zipper [17] visitor for any recursive type \(\mu_x F\) as a list of the derivative of that type, namely \(\mu_x \mu_x F = \mu_x F \times \mu_x \mu_x F\). In our case, calculating the derivative of the inductive tree, we get \(\mu_x 1 + (\mu_x \times \text{int} \times x) + (\mu_x \times \text{int} \times x)\), which corresponds to the visitor datatype.

\(^3\)Koka actually generalizes this using a multi-prompt delimited control monad that works for any control effect, with essentially the same principle.
2.7.2 Concurrent Execution. If multiple threads share a reference to a value, the reference count needs to be incremented and decremented using atomic operations which can be expensive. Ungar et al. [43] report slowdowns up to 50% when atomic reference counting operations are used. Nevertheless, in languages with unrestricted multi-threading, like Swift, almost all reference count operations need to assume that references are potentially thread-shared.

In Koka, the strong type system gives us additional guarantees about which variables may need atomic reference count operations. Following the solution of Ullrich and de Moura [42], we mark each object with whether it can be thread-shared or not, and supply an internal polymorphic operation \texttt{tshare : }\forall a. a \to \text{io }() \text{ which marks any object and its children recursively as being thread-shared. All objects start out as not thread-shared, and are only marked through explicit operations. In particular, when starting a new thread, the argument passed to the thread is marked as thread-shared. The only other operation that can cause thread sharing is setting a thread-shared mutable reference but this is quite uncommon in typical Koka code. The \texttt{drop} and \texttt{dup} operations can be implemented efficiently by avoiding atomic operations in the fast path by checking the thread-shared flag.

For example, \texttt{drop} may be implemented in C as:

```c
static inline void drop( block_t* b ) {
  if (atomic_dec(&b->header.rc) == 1) drop_check(b); // slow path
  else if (atomic_dec(&b->header.rc) == 1) drop_free(b);
}
```

However, this may still present quite some overhead as many \texttt{drop} operations are emitted.

In Koka we encode the reference count for thread-shared objects as a negative value. This enables us to use a \texttt{single} inlined test to see if we need to take the slow path for either a thread-shared object or an object that needs to be freed; and we can use a fast inlined path for the common case:

```c
static inline void drop( block_t* b ) {
  if (b->header.rc <= 1) drop_check(b); // slow path
  else b->header.rc--; // fast path
}
```

The \texttt{drop_check} function checks if the reference count is 1 to release it, or otherwise it adjusts the reference count atomically. We also use the negative values to implement a \texttt{sticky} range where very large reference counts stay without being further adjusted (preventing overflow, and keeping them alive for the rest of the program).

2.7.3 Mutation. Mutation in Koka is done through explicit mutable references. Here we look at first-class mutable reference cells, but Koka also has second-class mutable local variables that can be more convenient. A mutable reference cell is created with \texttt{ref}, dereferenced with \texttt{(!)} and updated using \texttt{(:=)}:

```plaintext
fun ref( init : a ) : st(h) ref(h,a)
fun (!)( r : ref(h,a) ) : st(h) a
fun (:=)( r : ref(h,a), x : a ) : st(h)()
```

where each operation has a stateful effect \texttt{st(h)} in some heap \texttt{h}. A reference cell of type \texttt{ref(h,a)} is a first-class \textit{value} that contains a reference to a value of type \texttt{a}. As such, there are always two reference counts involved: that of the reference itself, and that of value that is referenced.

When a mutable reference cell is thread-shared, this presents a problem as an update operation may \texttt{race} with a read operation to update the reference counts. The pseudocode implementation of both operations is:

```plaintext
fun (!)( r ) {
  val x = r->value
  x->value := x
}
fun (:=)( r, x ) {
  val y = r->value
  r->value := x
  drop y
}
```

The read operation \texttt{(!)} first reads the current reference in \texttt{x}, and then increments its reference count. Suppose though that before the \texttt{dup}, the thread is suspended and another thread writes to the same reference: it will read the same object into \texttt{y}, update the reference, and then drop \texttt{y} – and if \texttt{y} has a reference count of 1 it will be freed! When the other thread resumes, it will now try to \texttt{dup} the just-freed object.

To make this work correctly, we need to perform both operations atomically, either through a double-CAS [6], using hazard pointers [12, 30], or using some other locking mechanism. Either way, this can be quite expensive. Fortunately, in our setting, we can avoid the slow path in most cases. First of all, since FBIP allows for the efficiency of in-place updates with a purely functional specification (Section 2.6), we expect mutable references to be a last resort rather than the default. Secondly, as discussed in Section 2.7.2, we can also check if a mutable reference is actually thread-shared and thus avoid the atomic code path almost all of the time. In other settings though mutability can be costly; for example in Swift objects are mutable and behave like a mutable reference cell where most fields can be updated in-place, and therefore many field accesses need to be treated just like mutable reference cell operations. Moreover, as discussed in the previous section as well, the compiler must assume most of the time that objects might be thread-shared and thus use the slow atomic code path [43].

2.7.4 Cycles. A known limitation of reference counting is that it cannot release cyclic data structures. Just like with mutability, we try to mitigate its performance impact by reducing the potential for this to occur in the first place. In Koka, almost all data types are immutable and either \texttt{inductive} or \texttt{coinductive}. It can be shown that such data types are never cyclic (and functions that recurse over such data types always terminate).

---

\footnote{Since the thread-shared sign-bit is \textit{stable}, we can do the test \texttt{b->header.cr} == 1 without needing expensive atomic operations and can use a \texttt{memory_order_relaxed} atomic read.}
In practice, mutable references are the main way to construct cyclic data. Since mutable references are uncommon in our setting, we leave the responsibility to the programmer to break cycles by explicitly clearing a reference cell that may be part of a cycle. Since this strategy is also used by Swift, a widely used language where most object fields are mutable, we believe this is a reasonable approach to take for now. However, we have plans for future improvements: since we know statically that only mutable references are able to form a cycle, we could generate code that tracks those data types at run time and may perform a more efficient form of incremental cycle collection.

2.7.5 Summary. In summary, we have shown how static guarantees at compile-time can be used to mitigate the performance impact of concurrency and the risk of cycles. This paper does not yet present a general solution to all problems with reference counting and future work is required to explore how cycles can be handled more efficiently, and how well Perceus can be used with implicit control flow. Yet, we expect that our approach gives new insights in the general design space of reference counting, and showcase that precise reference counting can be a viable alternative to other approaches. In practice, we found that Perceus has good performance, which is discussed in Section 4.

3 A Linear Resource Calculus

In this section we present a novel linear resource calculus, $\lambda^1$, which is closely based on linear logic. The operational semantics of $\lambda^1$ is formalized in an explicit heap with reference counting, and we prove that the operational semantics is sound. We then formalize Perceus as a sound and precise syntax-directed algorithm of $\lambda^1$ and thus provide a theoretic foundation for Perceus.

3.1 Syntax

Figure 4 defines the syntax of our linear resource calculus $\lambda^1$. It is essentially an untyped lambda calculus extended with explicit binding as $\text{val } x = e_1; e_2$, and pattern matching as $\text{match } x \{ p_i \rightarrow e_i \}$. We assume all patterns in match are mutually exclusive, and all pattern binders are distinct. Syntactic constructs in gray are only generated in derivations of the calculus and are not exposed to users. Among those constructs, dup and drop form the basic instructions of reference counting.

Contexts $\Delta, \Gamma$ are multisets containing variable names. We use the compact comma notation for summing (or splitting) multisets. For example, $(\Delta, x)$ adds $x$ to $\Delta$, and $(\Gamma_1, \Gamma_2)$ appends two multisets $\Gamma_1$ and $\Gamma_2$. The set of free variables of an expression $e$ is denoted by $\text{fv}(e)$, and the set of bound variables of a pattern $p$ by $\text{bv}(p)$.

3.2 The Linear Resource Calculus

The derivation $\Delta \mid \Gamma \vdash e \rightarrow e'$ in Figure 5 reads as follows: given a borrowed environment $\Delta$, a linear environment $\Gamma$, an expression $e$ is translated into an expression $e'$ with explicit reference counting instructions. We call variables in the linear environment owned.

The key idea of $\lambda^1$ is that each resource (i.e., owned variable) is consumed exactly once. That is, a resource needs to be explicitly duplicated (in rule $\text{dup}$) if it is needed more than once; or be explicitly dropped (in rule $\text{drop}$) if it is not needed. The rules are closely related to linear typing.

Following the key idea, the variable rule $\text{var}$ consumes a resource when we own and only own $x$ exactly once in the owned environment. For example, to derive the $k$ combinator, $\lambda x. y. x$, we need to apply $\text{drop}$ to be able to discard $y$, which gives $\lambda x. \text{drop } y. x$.

The $\text{app}$ rule splits the owned environment $\Gamma$ into two separate contexts $\Gamma_1$ and $\Gamma_2$ for expression $e_1$ and $e_2$ respectively. Each expression then consumes its corresponding owned environment. Since $\Gamma_2$ is consumed in the $e_2$ derivation, we know that resources in $\Gamma_2$ are surely alive when deriving $e_1$, and thus we can borrow $\Gamma_2$ in the $e_1$ derivation. The rule is quite similar to the [$\text{let!}$] rule of Wadler’s linear type rules [44,pg.14] where a linear type can be “borrowed” as a regular type during evaluation of a binding.

Borrowing is important as it allows us to conduct a dup as late as possible, or otherwise we will need to duplicate enough resources before we can divide the owned environment. Consider $\lambda f. g. x. (f. x) (g. x)$. Without borrowing, we have to duplicate $x$ before the application, resulting in practice, mutable references are the main way to construct cyclic data. Since mutable references are uncommon in our setting, we leave the responsibility to the programmer to break cycles by explicitly clearing a reference cell that may be part of a cycle. Since this strategy is also used by Swift, a widely used language where most object fields are mutable, we believe this is a reasonable approach to take for now. However, we have plans for future improvements: since we know statically that only mutable references are able to form a cycle, we could generate code that tracks those data types at run time and may perform a more efficient form of incremental cycle collection.
applied, so it is derived under an empty borrowed environment only owning the argument and the free variables (in the closure). The translated lambda is also annotated with \(y_1\), as \(\lambda y_1 x. e\), so we know precisely the resources the lambda should own when evaluated in a heap semantics. We often omit the annotation when it is irrelevant.

The \textbf{bind} rule is similar to application and borrows \(\Gamma_2\) in the derivation for the bound expression. This is the main reason to not consider \(\text{val } x = e_1; e_2\) as syntactic sugar for \((\lambda x. e_2) e_1\). The \textbf{match} rule consumes the scrutinee and owns the bound variables in each pattern for each branch. For constructors (rule \textit{con}), we divide the owned environment into \(n\) parts for each component, and allow each component derivation to borrow the owned environment of the components derived later.

We use the notation \([e]\) to erase all drop and dup in the expression \(e\). We can now state that derivations leave expressions unchanged except for inserting \(\text{dup/drop} \) operations:

\[ \text{if } \Delta \mid \Gamma \vdash e \rightsquigarrow e' \text{ then } e = [e]'. \]

\textbf{Lemma 1.} (Translation only inserts dup/drop)

If \(\Delta \mid \Gamma \vdash e \rightsquigarrow e'\) then \(e = [e]'\).

\subsection{3.3 Semantics}

Figure 6 defines standard semantics for \(\lambda^1\) using strict evaluation contexts \cite{45}. The evaluation contexts uniquely determine where to apply an evaluation step. As such, evaluation contexts neatly abstract from the usual implementation context of a stack and program counter. Rule (\textit{match}) replies on the internal form of expression \textit{match} \( e \{ p_i \mapsto e_i' \} \); after substitution (\textit{app}), values may appear in positions where only variables were allowed, and this is exactly what enables us to do pattern match on a data constructor.

In Figure 7 we define our target semantics of a reference counted heap, so sharing of values becomes explicit and substitution only substitutes variables. Here, each heap entry \( x \rightarrow^n v \) points to a value \( v \) with a reference count of \( n \) (with \( n \geq 1 \)). In these semantics, values other than variables are allocated in the heap with rule (\textit{lam},) and rule (\textit{con},). The evaluation rules discard entries from the heap when the reference count drops to zero. Any allocated lambda is annotated as \(\lambda y_1 x. e\) to clarify that these are essentially \textit{closures} holding an environment \(y_1\) and a code pointer \(\lambda x. e\). Note that it is important that the environment \(y_1\) is a multi-set. After the initial translation, \(y_1\) will be equivalent to the free variables in the body (see rule \textit{lam}), but during evaluation substitution may substitute several variables with the same reference. To keep reference counts correct, we need to keep considering each one as a separate entry in the closure environment.

When applying an abstraction, rule (\textit{app},) needs to satisfy the assumptions made when deriving the abstraction in rule \textit{lam}. First, the (\textit{app},) rule inserts \textit{dup} to duplicate variables \(y_1\), as these are owned in rule \textit{lam}. It then drops the reference to the closure itself. Rule (\textit{match},) is similar to
We say a variable $x$ is reachable if for some $y$, we have $\text{reach}(y, x) \land y \mapsto^* v \in H$ and $\text{reach}(x, H \mid e)$.

With reachability, we can formally show:

$$\text{H} \mid e \mapsto_r \text{H'} \mid e' \quad [\text{EVAL}]$$

To prove this theorem, we need to maintain strong invariants.

Theorem 1. (Reference-counted heap semantics is sound)

If $\varnothing \ni e \mapsto e'$, then also $\varnothing \ni e' \mapsto^*$. For every intermediate state $H_i \mid e_i$, we have that for all $y \in \text{dom}(H_i)$, $\text{reach}(y, H_i \mid e_i)$.

In Appendix D.3, we further show that the reference counts are exactly equal to the number of actual references to the resource. Notably, to capture the essence of precise reference counting, $\lambda^1$ does not model mutable references (Section 2.7.3).

3.4 Perceus

Figure 8 defines syntax directed derivation $\tau_r$ for our resource calculus and as such specifies our Perceus algorithm. Like before, $\Delta \mid \Gamma \vdash_\tau e \mapsto e'$ translates an expression $e$ to $e'$ under an borrowed environment $\Delta$ and an owned environment $\Gamma$. During the derivation, we maintain the following invariants: (1) $\Delta \cap \Gamma = \varnothing$; (2) $\Gamma \subseteq \text{fv}(e)$; (3) $\text{fv}(e) \subseteq \Delta, \Gamma$; and (4) multiplicity of each member in $\Delta, \Gamma$ is 1. We ensure these properties hold by construction at any step in a derivation.
Perceus: Garbage Free Reference Counting with Reuse

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta \vdash \Gamma . t_3 \ e \leadsto e'$</td>
<td>$\Delta \cap \Gamma = \emptyset \quad \Gamma \subseteq \text{fv}(e) \quad \text{fv}(e) \subseteq \Delta, \Gamma$ multiplicity of each member in $\Delta, \Gamma$ is 1</td>
</tr>
</tbody>
</table>

![Fig. 8. Syntax-directed linear resource rules of $\lambda^i$.](image)

The Perceus rules are set up to do precise reference counting: we delay a dup operation to come as late as possible, pushing them out to the leaves of a derivation; and we generate a drop operation as soon as possible, right after a binding or at the start of a branch.

Rule svar-dup borrows $x$ by inserting a dup. The sapp rule now deterministically finds a good split of the environment $\Gamma$. We pass the intersection of $\Gamma$ with the free variables in $e_2$ to the $e_2$ derivation. Otherwise the rule is the same as in the declarative system. For abstraction and binding we have two variants: one where the binding is actually in the free variables of the expression (rule slam and sbind), and one where the binding can be immediately dropped as it is unused (rule slam-drop and sbind-drop). In the abstraction rule, we know that $\Gamma \subseteq \text{fv}(\lambda x. e)$ and thus $\Gamma \subseteq ys$. If there are any free variables not in $\Gamma$, they must be part of the borrowed environment (as $\Delta$) and these must be duplicated to ensure ownership. The bind rules are similarly constructed as a mixture of sapp and slam.

The smatch rule is interesting as in each branch there may be variables that can be dropped as they no longer occur as free variables in that branch. The owned environment $\Gamma_i$ in the $i$th branch is the intersection of $(\Gamma_i, \text{bv}(p_i))$ and the free variables in that branch; any other owned variables (as $\Gamma_i'$) are dropped at the start of the branch. Rule scon deterministically splits the environment $\Gamma$ as in rule sapp.

We show that the Perceus algorithm is sound by showing that for each rule there exists a derivation in the declarative linear resource calculus. The proof is given in Appendix D.4.

**Theorem 3.** (Syntax directed translation is sound.) If $\Delta \vdash \Gamma . t_3 \ e \leadsto e'$ then also $\Delta \vdash \Gamma \vdash e \leadsto e'$.

More importantly, we prove that any translation resulting from the Perceus algorithm is precise, where any intermediate state in the evaluation is garbage free (Appendix D.5):

**Theorem 4.** (Perceus is precise and garbage free) If $\emptyset \vdash \emptyset \vdash e \leadsto e'$ and $\emptyset \vdash e' \rightarrow^*_{\text{lam}} H \vdash x$, then for every intermediate state $H_i \mid e_i$ that is not at a dup/drop operation ($e_i \neq E[\text{drop } x; e'_i]$ and $e_i \neq E[\text{dup } x; e_i'])$, we have that for all $y \in \text{dom}(H_i)$, reach$(y, H_i \mid e_i)$.

This theorem states that after evaluating any immediate reference counting instructions, every variable in the heap is reachable from the erased expression. This rules out, for example, $y \mapsto (\lambda x. x) \ (\text{drop } y; (\text{drop } y))$. As $y$ is not in the free variables of the erased expression. Just like Theorem 2, if the system is extended with mutable references, then Theorem 4 could be generalized such that every resource is either reachable from the erased expression, or it is part of a cycle.

The implementation of Perceus is further extended with the optimizations described in Section 2. As the component transformations, including inlining and dup/drop fusion, are standard, the soundness of those optimizations follows naturally and a proof is beyond the scope of this paper.
We also run Koka “no-opt” with reuse analysis and drop/reuse specialization disabled to measure the impact of those optimizations.

- OCaml 4.08.1. This has a stop-the-world generational collector with a minor and major heap. The minor heap uses a copying collector, while a tracing collector is used for the major heap [8, 31, Chap. 22]. The Koka benchmarks correspond essentially one-to-one to the OCaml versions.
- Haskell, GHC 8.6.5. A highly optimizing compiler with a multi generational garbage collector. The benchmark sources again correspond very closely, but since Haskell has lazy semantics, we used strictness annotations in the data structures to speed up the benchmarks, as well as to ensure that the same amount of work is done.
- Swift 5.3. The only other language in this comparison where the compiler uses reference counting [4, 43]. The benchmarks are directly translated to Swift in a functional style without using direct mutation. However, we translated tail-recursive definitions to explicit loops with local variables.
- Java SE 15.0.1. Uses the HotSpot JVM and the G1 concurrent, low-latency, generational garbage collector. The benchmarks are directly translated from Swift.
- C++, gcc 9.3.0. A highly optimizing compiler with manual memory management. Without automatic memory management, many benchmarks are difficult to express directly in C++ as they use persistent and shared data structures. To implement these faithfully would essentially require manual reference counting. Instead, we use C++ as our performance baseline: if provided, we either use in-place updates without supporting persistence (as in rbtree which uses std::map) or we do not reclaim memory at all (as in deriv, nqueens, and cfold).

The benchmarks are all chosen to be medium sized and non-trivial, and all stress memory allocation with little computation. Most of these are based on the benchmark suite of Lean [42] and all are available in the Koka repository [1]. The execution times and peak working set averaged over 10 runs and normalized to Koka are given in Figure 9. When a benchmark is not available for a particular language, it is marked as × in the figures.

- rbtree: inserts 42 million items into a red-black tree.
- rbtree-ck: a variant of rbtree that keeps a list of every 5th subtree and thus shares many subtrees.
- deriv: the symbolic derivative of a large expression.
- nqueens: calculates all solutions for the n-queens problem of size 13 into a list, and returns the length of that list. The solution lists share many sub-solutions.
- cfold: constant-folding over a large symbolic expression.

We can see from Figure 9 that even though Koka has few optimizations besides the reference counting ones, it performs very well compared to these mature systems, often outperforming by a significant margin – both in execution time and peak working set. We only discuss the overall results...
here, but appendix B includes a detailed discussion of each individual benchmark.

In the rbtree benchmark, the functional implementation of Koka is within 10% of the highly optimized, in-place updating, std::map implementation in C++. We believe this is partly because C++ allocations must be 16-byte aligned while the Koka allocator can use 8-byte alignment and thus allocate a bit less. The rbtree benchmark also shows the potential effectiveness of the reference count optimizations where the "no-opt" version is more than 2× slower. However, in benchmarks with lots of sharing, like deriv and qknees, the optimizations are less effective.

Since Perceus is garbage free we would expect that Koka always uses less memory than a GC based system. This is indeed the case in our benchmarks – except for OCaml in deriv. Through manual inspection of OCaml’s machine code, we believe that OCaml avoids some allocations by applying "case of case" transformations [35] which are not (yet) available in the Koka compiler.

5 Related Work

Our work is closely based on the reference counting algorithm in the Lean theorem prover as described by Ullrich and de Moura [42]. They describe reuse analysis based on reset/reuse instructions, and describe both reference counting based on ownership (i.e. precise) but also support borrowed parameters. We extend their work with drop- and reuse specialization, and generalize to a general purpose language with side-effects and complex control flow. We also introduce a novel formalization of reference counting with the linear resource calculus, and define our algorithm in terms of that. As such, the Perceus algorithm may differ from the Lean one as that is specified over a lower-level calculus that uses explicit partial application nodes (pap) and has no first-class lambda expressions.

Schulte [39] describes an algorithm for inserting reference count instructions in a small first-order language and shows a limited form of reuse analysis, called “reusage” (transformation T14). Using explicit reference count instructions in order to optimize them via static analysis is described as early as Barth [2]. Mutating unique references in place has traditionally focused on array updates [16], as in functional array languages like Sisal [29] and SaC [13, 38]. Férey and Shankar [9] provide functional array primitives that use in-place mutation if the array has a unique reference; we plan to add these to Koka. We believe this would work especially well in combination with reuse-analysis for BTree-like structures using trees of small functional arrays.

The λ¹ calculus is closely based on linear logic. The main difference is that, in systems with linear types (e.g., Wadler [44]), the linear (or uniqueness) property is static whereas reference counts track this dynamically. Turner and Wadler [41] give a heap-based operational interpretation which does not need reference counts as linearity is tracked by the type system. In contrast, Chirimar et al. [3] give an interpretation of linear logic in terms of reference counting, but in their system, values with a linear type are not guaranteed to have a unique reference at runtime.

The Swift language is widely used in iOS development and uses reference counting with an explicit representation in its intermediate language. There is no reuse analysis but, as remarked by Ullrich and de Moura [42], this may not be so important for Swift as typical programs mutate objects in-place. There is no cycle collection for Swift, but despite the widespread usage of mutation this seems to be not a large problem in practice. Since it can be easy to create accidental cycles through the self pointer in callbacks, Swift has good support for weak references to break such cycles in a declarative manner.

Swift uses reference counting with an explicit representation in its intermediate language. Ungar et al. [43] optimize atomic reference counts by tagging objects that can be potentially thread-shared. Later work by Choi et al. [4], uses biased reference counting to avoid many atomic updates.

Another recent language that uses reference counting is Nim. The reference counting method is scope-based and uses non-atomic operations (and objects cannot be shared across threads without extra precautions). Nim can be configured to use ORC reference counting which extends the basic ARC collector with a cycle collection [48]. Nim has the acyclic annotation to identify data types that are (co)-inductive, as well as the (unsafe) cursor annotation for variables that should not be reference counted.

In our work we focus on precise and garbage free reference counting which enables static optimization of reference count instructions. On the other extreme, Deutsch and Bobrow [7] consider deferred reference counting – any reference count operations on stack-based local variables are deferred and only the reference counts of fields in the heap are maintained. Much like a tracing collector, the stack roots are periodically scanned and deferred reference counting operations are performed. Levanoni and Petrunk [25] extend this work and present a high performance reference counting collector for Java that uses the sliding view algorithm to avoid many intermediate reference counting operations and needs no synchronization on the write barrier.

6 Conclusion

In this paper we present Perceus, a precise reference counting system with reuse and specialization, which is built upon λ¹, a novel linear resource calculus closely based on linear logic. Our full implementation in Koka is competitive with other mature memory collectors. In future work, we would like to study ways to handle cycles efficiently. Moreover, we would like to integrate selective “borrowing” into Perceus – this would make certain programs no longer be garbage...
free, but we believe it could deliver further performance improvements if judiciously applied.

7 Acknowledgements

We like to thank Erez Petrank for the discussions on the race conditions that can occur with in-place updates and reference counts.

References

Appendix

A Red-black tree implementation

Figure 10 shows balanced insertion into a red-black tree using Okasaki’s algorithm [34].

B Extended Benchmark Discussion

Here we discuss the results of each individual benchmark from Figure 9 in more detail:

- **rbtree**: this performs 42 million insertions into a red-black balanced tree and after that folds over the tree counting the true elements. Here the reuse analysis of Koka shines (as shown in Section 2.4) and it outperforms all other systems where only OCaml is close in performance — rebalancing generates lots of short-lived object allocation which are a great fit for the minor heap copying-collector of OCaml with fast aggregated bump-pointer allocation. The C++ benchmark is implemented using the in-place updating `std::map` implementation, which internally uses a highly optimized red-black tree implementation [10]. Surprisingly, the purely functional Koka implementation is within 10% of the C++ performance. Since the insertion operations are the same, we believe this is partly because C++ allocations must be 16-byte aligned while the Koka allocator can use 8-byte alignment in the allocations and thus allocate a bit less (as apparent in Figure 9) (and similarly, bump pointer allocation in OCaml can be faster than general `malloc/free`). Java performs close to C++ here but also uses almost 10× the memory of Koka (1.7GiB vs. 170MiB, Figure 9). This can be reduced to about 1.5× by providing tuning parameters on the command line but that also made it slower on our system. The effects of the reuse analysis and specialization optimizations can also be significant with optimized Koka being more than twice as fast as "no-opt".

- **rbtree-ck**: in previous reviews, it has been suggested that rbtree is biased to reference counting as it has no shared subtrees and thus reuse analysis can use in-place updates all the time. The rbtree-ck benchmark remedies this and is a variant of rbtree that keeps a list of every 5th tree generated and thus shares many subtrees. Again, Koka outperforms all other systems. Haskell and OCaml are now relatively slower than in rbtree — we conjecture this is due to extra copying between generations, and perhaps due to increased tracing cost.

- **deriv**: calculates the derivative of a large symbolic expression. Again, Koka does very well here. Interestingly, the memory usage of OCaml is slightly less than Koka — since Perceus is garbage free we would expect that Koka always uses less memory than a GC based system. From studying the generated code of OCaml we believe that it is because the optimizing OCaml compiler can avoid some allocations by applying "case of case" transformations [35] which the

type color {
  Red
  Black
}
type tree {
  Leaf
  Node(color:color, left:tree, key:int, value:bool, right:tree)
}

fun is-red(t : tree) : bool {
  match(t) {
    Node(Red) -> True
    _ -> False
  }
}

fun bal-left(l:tree, k: int, v: bool, r: tree): tree {
  match(l) {
    Leaf -> Leaf
    Node(_, Node(Red, lx, kx, vx, rx), ky, vy, ry) ->
      Node(Red, Node(Black, lx, kx, vx, rx), ky, vy, Node(Black, ry, k, v, r))
    _ -> Node(Red, Node(Black, ly, ky, vy, ry))
  }
  Node(_, lx, kx, vx, Node(Red, ly, ky, vy, ry))
  Node(_, lx, kx, vy, Node(Red, ly, ky, vx, ry))
  Node(_, lx, kx, vy, Node(Red, ly, vx, ry))
  Node(_, lx, kx, vx, Node(Black, l, k, v, rx))
  Node(_, lx, kx, vx, Node(Black, l, k, v, ry))
  Node(_, lx, kx, vx, Node(Black, l, k, vx, rx))
  Node(_, lx, kx, vx, Node(Black, l, k, vx, ry))
}

fun bal-right(l: tree, k: int, v: bool, r: tree): tree {
  match(r) {
    Leaf -> Leaf
    Node(_, Node(Red, l, kx, vx, rx), ky, vy, ry) ->
      Node(Red, Node(Black, l, k, v, lx), kx, vx, Node(Black, rx, k, v, r))
    _ -> Node(Red, Node(Black, l, k, v, rx))
  }
  Node(_, l, kx, vx, Node(Red, l, kx, rx), k, v, r)
  Node(_, l, kx, vx, Node(Red, l, kx, ry), k, v, r)
  Node(_, l, kx, vy, Node(Red, l, kx, ry), k, v, r)
  Node(_, l, kx, vy, Node(Red, l, kx, rx), k, v, r)
  Node(_, l, kx, vy, Node(Red, l, kx, ry), k, v, r)
  Node(_, l, kx, vy, Node(Red, l, kx, rx), k, v, r)
  Node(_, l, kx, vy, Node(Red, l, kx, ry), k, v, r)
  Node(_, l, kx, vy, Node(Red, l, kx, rx), k, v, r)
  Node(_, l, kx, vy, Node(Red, l, kx, ry), k, v, r)
}

fun set-black(t: tree) : tree {
  match(t) {
    Node(_, l, k, vx, rx) -> Node(Black, l, k, vx, rx)
    _ -> t
  }
}

fun insert(t: tree, k: int, v: bool): tree {
  if (is-red(t))
    then set-black(ins(t, k, v))
  else ins(t, k, v)
}

Fig. 10. Red-black tree balanced insertion in Koka

naive Koka compiler is not (yet) doing. It is also interesting to see that the "no-opt" Koka is only a bit slower than
optimized Koka here. This is probably due to the sharing of many sub-expressions when calculating the derivative. This in turn causes the code resulting from drop/reuse specialization and reuse analysis to mostly use the “slow” path which is equivalent to the one in “no-opt”. Finally, Java performs best on this benchmark; we can see while running the benchmark that it can run the G1 collector fully concurrent on another core.

- \textit{nqueens}: calculates all solutions for the n-queens problem of size 13 into a list, and returns the length of that list. The solution lists share many sub-solutions and, as in \textit{deriv}, for the C++ version we do \textit{not} free any memory (but do allocate the same objects as the other benchmarks). Again, Koka is quite competitive even with the large amount of shared structures, and the peak working set is significantly lower.

- \textit{cfold}: performs constant-folding over a large symbolic expression. This benchmark is similar to the \textit{deriv} benchmark and manipulates a complex expression graph. Koka does significantly better than other systems. Just as in \textit{deriv}, we see that OCaml uses slightly less memory as it can avoid some allocations by optimizing well. The “no-opt” version of Koka also uses 8% less memory; this is because the reuse analysis essentially holds on to memory for later reuse. Just like with scoped based reference counting that may lead to increased memory usage in some situations.

An interesting overall observation is that the reference counting implementation of Swift is not doing as well as Koka – this may be partly due to the language and compiler, but we also believe that this is a confirmation of our initial hypothesis where we argue that a combination of static compiler optimizations with dynamic runtime checks (e.g. \textit{is-unique}) are needed for best results. As discussed for example in Section 2.7.2, some of the optimizations we perform are difficult to do in Swift as the static guarantees of the language are not strong enough.

C Further Benchmarks

Figure 11 show execution time and peak working sets on a 10-core Intel Core i9-7900X at 3.30GHz, Ubuntu 20.04.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig11}
\caption{Relative execution time and peak working set with respect to Koka (lower is better). On a 10-core Intel Core i9-7900X at 3.30GHz, Ubuntu 20.04.}
\end{figure}
D Proofs

D.1 A Heap Reference Counting Calculus

\[
\dfrac{H \vdash v_1 \vdash H_1 \ldots H_{n-1} \vdash v_n \vdash H_n}{H \vdash C \; v_1 \ldots v_n \vdash H_n} \quad \text{[DRCON]}
\]

\[
\dfrac{H, x \mapsto^{n+1} v \vdash x \vdash H, x \mapsto^n v}{H \vdash ys \vdash H_1} \quad \text{[DRVAR]}
\]

\[
\dfrac{H, x \mapsto^n \lambda y z . e \vdash x \vdash H_1}{H \vdash ys \vdash H_1} \quad \text{[DRVARLAM]}
\]

\[
\dfrac{H, x \mapsto^n C \; ys \vdash x \vdash H_1}{H \vdash ys \vdash H_1} \quad \text{[DRVARCON]}
\]

\[
\dfrac{H, x \mapsto^n v \vdash e \vdash H_1}{H, x \mapsto^n v \vdash e \vdash H_1} \quad \text{[DRDUP]}
\]

\[
\dfrac{H, x \mapsto^n v \vdash e \vdash H_1}{H, x \mapsto^n v \vdash e \vdash H_1} \quad \text{[DRDROP]}
\]

\[
\dfrac{H \vdash drop \; ys; e \vdash H_1}{H \vdash drop \; ys; e \vdash H_1} \quad \text{[DRDROPCON]}
\]

\[
\dfrac{H \vdash drop \; ys; e \vdash H_1}{H \vdash drop \; ys; e \vdash H_1} \quad \text{[DRDROPLAM]}
\]

\[
\dfrac{H \vdash ys \vdash H_1}{H \vdash ys \vdash H_1} \quad \text{[DRLAM]}
\]

\[
\dfrac{H \vdash e_1 \vdash H_1 \quad H \vdash e_2 \vdash H_2}{H \vdash e_1 \ldots e_2 \vdash H_1 \ldots H_2} \quad \text{[DRAPP]}
\]

\[
\dfrac{H \vdash e_1 \vdash H_1 \quad H_1, x \mapsto^\top () \vdash e_2 \vdash H_2 \quad x \notin H, H_2}{H \vdash val \; x = e_1; e_2 \vdash H_1} \quad \text{[DRBIND]}
\]

\[
\dfrac{H \vdash x \vdash H_1 \quad H_1, \left[ \text{bv}(p_i) \right] \vdash e_i \vdash H' \quad \text{bv}(p_i) \notin H, H'}{H \vdash \text{match} \; x \{ p_i \rightarrow e_i \} \vdash H'} \quad \text{[DRMATCH]}
\]

Lemma 2. (Heap Reference Counting Free variables)
If \( H_1 \vdash e \vdash H_2 \), then \( \text{fv}(e) \in H_1 \), and \( \text{fv}(H_2) \in H_1 \) with same domains.

Proof. (Of Lemma 2) By a straightforward induction on the rules. \( \Box \)

Definition 2. (Extension)
H is extended with \( x \), denoted as \( H \# x \), where \# works as follows:
1. if \( H = H' \), \( x \mapsto^n v \), then \( H \# x = H' \), \( x \mapsto^{n+1} ys \);
2. if \( x \notin H \), then \( H \# x = (H, x \mapsto^1 v) \# \text{fv}(v) \).

We omit the domain of \( x \) in \( H \# x \) for simplicity. The domain should always be available by inspecting the heap (in (1)) or via explicit passing (in (2)).
We only focus on situations where there is no cycles in the dependency of $x$ (but we are fine with existing cycles in $H$), so that the extension terminates. That implies $(H, x \mapsto\triangleright v) \not\triangleright f(v) = H \triangleright f(v), x \mapsto\triangleright v$ in (2).

**Lemma 3.** *(Drop is dual to extension)*

If $H_1 \vdash \text{drop } x; () \vdash H_2$, then $H_1 = H_2 \triangleright x$. Similarly, if $H_1 \vdash x \vdash H_2$, then $H_1 = H_2 \not\triangleright x$.

**Proof. (Of Lemma 3)** By induction on the judgment.

   case

   $H, x \mapsto^{n+1} ys \vdash \text{drop } x; () \vdash H, x \mapsto^n ys$  
   \textbf{DBDROP, DBCON}

   case

   $H, x \mapsto^1 \lambda y z. e \vdash \text{drop } x; () \vdash H_1$  
   \textbf{given}

   $H \vdash \text{drop } ys; () \vdash H_1$  
   \textbf{DBDROPLAM}

   $H = H_1 \not\triangleright ys$  
   \textbf{I.H.}

   $H, x \mapsto^1 \lambda y z. e = H_1 \not\triangleright x$  
   \textbf{by definition}

   $\square$

**Lemma 4.** *(Extension is dual to drop)*

$H \not\triangleright x \vdash \text{drop } x; () \vdash H$. Similarly, $H \not\triangleright x \vdash x \vdash H$.

**Proof. (Of Lemma 4)** By induction on $\not\triangleright x$.

   case

   $H = H^\prime, x \mapsto^n ys$  
   \textbf{if}

   $H \not\triangleright x = H^\prime, x \mapsto^{n+1} ys$  
   \textbf{by definition}

   $H^\prime, x \mapsto^{n+1} ys \vdash \text{drop } x; () \vdash H$  
   \textbf{DBDROP}

   case

   $x \not\in H$  
   \textbf{if}

   $ys = f(v)$  
   \textbf{let}

   $H \not\triangleright x = H \not\triangleright ys, x \mapsto^1 v$  
   \textbf{by definition}

   $H \not\triangleright ys \vdash ys \vdash H$  
   \textbf{I.H.}

   $H \not\triangleright ys, x \mapsto^1 v \vdash x \vdash H$  
   \textbf{DRVARLAM or DRVARCON}

   $\square$

**Lemma 5.** *(Extension Commutativity)*

$H \not\triangleright x \not\triangleright y = H \not\triangleright y \not\triangleright x$.

**Proof. (Of Lemma 5)** By induction on $\not\triangleright x$ and $\not\triangleright y$, then we do case analysis. case $x \in H$. Then $H = H^\prime, x \mapsto^n v$.

   By definition, $H \not\triangleright x \not\triangleright y = (H^\prime, x \mapsto^{n+1} v) \not\triangleright y$. Since the way $\not\triangleright$ works only depends on whether $x$ exists but not the exact number of its occurrence, we can decrease the number of $x$, do $\not\triangleright y$ and then add $x$ back. That is, $(H^\prime, x \mapsto^{n+1} v) \not\triangleright y = (H^\prime, x \mapsto^{n} v) \not\triangleright y \not\triangleright x = H \not\triangleright y \not\triangleright x$.

   case $y \in H$ is similar as the previous case.

   case $x, y \not\in H$. Then $H \not\triangleright x = H \not\triangleright xs, x \mapsto^1 v$ where $xs = f(v)$.

   subcase Assume $\not\triangleright y$ won’t cause $\not\triangleright x$, then $\not\triangleright y$ doesn’t care about the existence of $x$.

   So $(H \not\triangleright xs, x \mapsto^1 v) \not\triangleright y = H \not\triangleright xs \not\triangleright y, x \mapsto^1 v$

   $= H \not\triangleright y \not\triangleright xs, x \mapsto^1 v$ by I.H.

   $= H \not\triangleright y \not\triangleright x$ by definition.
subcase  Or otherwise $+y$ will cause $+x$. Since there is no cycle in the dependency, that means $+x$ won’t cause $+y$. Then we can prove it as in the previous case. □

D.1.1 Relating to linear resource calculus.

**Definition 3. (Context to Dependency Heap)**

Given a context $\Gamma$, $[\Gamma]$ defines a dependency heap, with all $x$ becoming $x \mapsto n$ if $x$ appears $n$ times in $\Gamma$.

**Lemma 6.**

$[\Gamma, x] \vdash x + [\Gamma]$. Similarly, if $[\Delta] \vdash e \vdash H$, then $[\Gamma, x] \vdash \text{drop } x; e \vdash H$.

**Proof.** The goal holds by rule $\text{drvar (drdrop)}$ when $x \in \Gamma$ or by rule $\text{drvarcon (drdropcon)}$ if $x \not\in \Gamma$. □

**Lemma 7. (linear resource calculus relates to reference counting)**

If $\Delta \mid \Gamma \vdash e \Rightarrow e'$, then $[\Delta, \Gamma] \vdash e' \vdash [\Delta]$.

**Proof. (Of Lemma 7)** By induction on the elaboration.

case

$\Delta \mid x \vdash x \Rightarrow x$ given

$[[\Delta], x] \vdash x \vdash [[\Delta]]$ Lemma 6

case

$\Delta \mid \Gamma \vdash e \Rightarrow \text{dup } x; e'$ given

$\Delta \mid \Gamma, x \vdash e \Rightarrow e'$ given

$x \in \Delta, \Gamma$

$[[\Delta, \Gamma], x] \vdash e' \vdash [[\Delta]]$ I.H.

$[[\Delta, \Gamma]] \vdash \text{dup } x; e' \vdash [[\Delta]]$ $\text{drdup}$

case

$\Delta \mid \Gamma, x \vdash e \Rightarrow \text{drop } x; e'$ given

$\Delta \mid \Gamma \vdash e \Rightarrow e'$ given

$[[\Delta, \Gamma]] \vdash e' \vdash [[\Delta]]$ I.H.

$[[\Delta, \Gamma], x] \vdash \text{drop } x; e' \vdash [[\Delta]]$ Lemma 6

case

$\Delta \mid \Gamma \vdash \lambda x. e \Rightarrow \lambda y s. x. e'$ given

$\emptyset \mid y s, x \vdash e \Rightarrow e'$ given

$y s = \text{fv(}\lambda x. e\text{)}$ given

$[[\Delta, y s]] \vdash y s \vdash [[\Delta]]$ Lemma 6

$[[y s, x]] \vdash e' \vdash \emptyset$ I.H.

$[[\Delta, y s]] \vdash \lambda y s. x. e \vdash [[\Delta]]$ $\text{drlam}$

case

$\Delta \mid \Gamma_1, \Gamma_2 \vdash e_1 e_2 \Rightarrow e_1' e_2'$ given

$\Delta, \Gamma_1 \mid \Gamma_1 \vdash e_1 \Rightarrow e_1'$ given

$\Delta \mid \Gamma_2 \vdash e_2 \Rightarrow e_2'$ given

$[[\Delta, \Gamma_1, \Gamma_2]] \vdash e_1' \vdash [[\Delta, \Gamma_2]]$ I.H.

$[[\Delta, \Gamma_2]] \vdash e_2' \vdash [[\Delta]]$ I.H.

$[[\Delta, \Gamma_1, \Gamma_2]] \vdash e_1' e_2' \vdash [[\Delta]]$ $\text{drapp}$

case
Δ | Γ, x ⊢ val x = e_1; e_2 ⊸ val x = e'_1; e'_2 \quad \text{given}
Δ, Γ | x ⊢ e_1 ⊸ e'_1 \quad \text{given}
Δ | Γ, x ⊢ e_2 ⊸ e'_2 \quad \text{given}
x \not\in Δ, Γ, 1
\{Δ, Γ, 1\} + e'_1 + \{Δ, Γ\} \quad \text{I.H.}
\{Δ, 1\} + e'_2 + \{Δ\} \quad \text{I.H.}
\{Δ, Γ\} + \text{match } x \{p_i \mapsto e_i\} \quad \text{match } x \{p_i \mapsto e'_i\} \quad \text{DRBIND}
\text{case}
\Delta | Γ, x ⊢ \text{match } x \{p_i \mapsto e_i\} \quad \text{match } x \{p_i \mapsto e'_i\} \quad \text{given}
\Delta | Γ, \text{bv}(p_i) ⊢ e_i \mapsto e'_i \quad \text{given}
\{Δ, Γ, x\} + x + \{Δ, Γ\} \quad \text{Lemma 6}
\{Δ, Γ\} + \text{match } x \{p_i \mapsto e'_i\} + \{Δ\} \quad \text{I.H.}
\text{case}
\Delta | Γ, x ⊢ C v_1 \ldots v_n \rightarrow C v'_1 \ldots v'_n \quad \text{given}
\Delta, Γ_{i+1} \ldots Γ_n | Γ_i \vdash v_i \quad \text{given}
\{Δ, Γ_i, Γ_{i+1}, \ldots Γ_n\} ⊢ v_i \rightarrow \{Δ, Γ_i, Γ_{i+1}, \ldots Γ_n\} \quad \text{I.H.}
\{Δ, Γ_i, \ldots Γ_n\} ⊢ C v_1 \ldots v_n + \{Δ\} \quad \text{DRCON}
\square

D.1.2 Weakening.

Lemma 8. (Weakening)
If H_1 \vdash e \mapsto H_2, then H_1 + x \vdash e \mapsto H_2 + x.

Proof. (Of Lemma 8) By induction on the judgment.
\text{case}
H \vdash C v_1 \ldots v_n - | H_n \quad \text{given}
H \vdash v_i + H_i \quad \text{DRCON}
H + x \vdash v_i + H + x \quad \text{I.H.}
H + x \vdash C v_1 \ldots v_n + H + x \quad \text{DRCON}
\text{case}
H \vdash y + H_2 \quad \text{given}
H = H_2 + y \quad \text{Lemma 3}
H + x = H_2 + y + x
= H_2 + x + y \quad \text{Lemma 5}
H_2 + x + y \vdash y + H_2 + x \quad \text{Lemma 4}
\text{case}
H, y \mapsto^n ys \vdash \text{dup } y; e \vdash H_1 \quad \text{given}
H, y \mapsto^{n+1} ys \vdash e \vdash H_1 \quad \text{DRDUP}
(H, y \mapsto^{n} ys) + y \vdash e \vdash H_1 \quad \text{definition of +}
(H, y \mapsto^{n} ys) + y + x \vdash e \vdash H_1 + x \quad \text{I.H.}
(H, y \mapsto^{n} ys) + y + x
= (H, y \mapsto^{n} ys) + x + y \quad \text{Lemma 5}
(H, y \mapsto^{n} ys) + x + y \vdash e \vdash H_1 + x \quad \text{By substitution}
(H, y \mapsto^{n} ys) + x \vdash \text{dup } y; e \vdash H_1 + x \quad \text{DRDUP}
\text{case}
H ⊢ \text{drop } y; e ⊢ H_2 \quad \text{given}
H ⊢ \text{drop } y; () ⊢ H_3 \quad \text{follows}
H_3 ⊢ e ⊢ H_2 \quad \text{above}
H = H_3 + y \quad \text{Lemma 3}
H ⊢ x = H_3 + y + x
= H_3 + x + y \quad \text{Lemma 5}
H_3 + x + y ⊢ \text{drop } y; () ⊣ H_2 \quad \text{Lemma 4}
H_3 + x + y ⊢ e ⊢ H_2 + x \quad \text{I.H.}
H + x ⊢ \text{drop } y; e ⊢ H_2 + x \quad \text{Follows}
\quad \text{By substitution}
\text{case}

H ⊢ \lambda ys. z. e + H_1 \quad \text{given}
H ⊢ ys → H_1 \quad \text{given}
ys \mapsto 1 (,), z \mapsto 1 () + e + \emptyset \quad \text{given}
H ⊢ x + ys → H_1 + x \quad \text{L.H.}
H ⊢ x + \lambda z. e + H_1 + x \quad \text{drlam}
\quad \text{case}

H ⊢ e_1 e_2 + H_2 \quad \text{given}
H ⊢ e_1 + H_1 \quad \text{given}
H_1 ⊢ e_2 + H_2 \quad \text{given}
H ⊢ x + e_1 + H_1 + x \quad \text{I.H.}
H_1 + x + e_2 + H_2 + x \quad \text{I.H.}
H ⊢ x + e_1 e_2 + H_2 + x \quad \text{drapp}
\quad \text{case}

H ⊢ \text{val } z = e_1 : e_2 + H_2 \quad \text{given}
H ⊢ e_1 + H_1 \quad \text{given}
H_1, z \mapsto 1 () + e_2 + H_2 \quad \text{given}
z \notin H \quad \text{given}
H ⊢ x + e_1 + H_1 + x \quad \text{I.H.}
(H_1, z \mapsto 1 ()) + x + e_2 + H_2 + x \quad \text{I.H.}
z \notin H_1 \quad \text{Lemma 2}
(H_1, z \mapsto 1 ()) = H_1 + z
(H_1, z \mapsto 1 ()) + x = H_1 + z + x
= H_1 + x + z \quad \text{Lemma 5}
= H_1 + x, z \mapsto 1 ()
H_1 + x, z \mapsto 1 () ⊢ e_2 + H_2 + x \quad \text{by substitution}
H ⊢ x + e_1 e_2 + H_2 + x \quad \text{drbind}
\quad \text{case}
Proof

\[
\begin{align*}
\text{Lemma 9.} & \quad \text{(Reference counting semantics is sound (small step))} \\
& \text{If } e_1 \rightarrow e_2 \text{ and } H_1 \text{ ok, and } e'_1 \text{ ok, and } [H_1] e'_1 = e_1, \text{ and } H_1 \vdash e'_1 \vdash H'_1, \text{ then there exists } H_2, e'_2 \text{ such that } H_1 \vdash e'_1 \rightarrow^*_{r} H_2 \vdash e'_2, \text{ and } [H_2] e'_2 = e_2. \\
\text{Proof. (Of Lemma 9) By induction on the evaluation judgment.} \\
\text{case } (\text{app}) \ (\lambda x. e) \ v \rightarrow e[x:=v] \\
& [H_1] e'_1 = (\lambda x. e) \ v \quad \text{given} \\
& \quad \text{subcase } e'_1 = f \ z. \\
& \quad [H_1](f \ z) = (\lambda x. e) \ v \\
& \quad [H_1] f = (\lambda x. e) \\
& \quad [H_1] z = v \\
& \quad f \mapsto^n \lambda y. s \
& \quad e' \in H_1 \\
& \quad f \mapsto^n \lambda y. s \
& \quad e' \in H_1 \\
& \quad [H_1] e' = e \\
& \quad H_1 \vdash f \ z \vdash H_1' \\
& \quad f \mapsto^n \lambda y. s \
& \quad e' \in H_1 \\
& \quad [H_1] e' \in H_1 \\
& \quad H_1 \text{ ok} \\
& \quad ys \in H_1 \\
& \quad f \mapsto^n \lambda y. s \
& \quad e' \in H_1 \\
& \quad e'[x:=z] \\
& \quad H_1 \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad H_1 + ys \vdash e'[x:=z] \\
& \quad e'[x:=v] \\
& \quad \text{subcase } e'_1 = (\lambda x. e') \ z. 
\end{align*}
\]
\[
[H_1]((\lambda x. e') z) = (\lambda x. e) v
\]
given

\[
[H_1]e' = e
\]
follows

\[
[H_1]z = v
\]
follows

\[H \vdash (\lambda x. e') z \rightarrow H, f \rightarrow^1 \lambda y^x. e' \mid f z\]
\[(\text{lam})\] and step

\[f \text{ fresh}, \ y's = \text{fv}(\lambda x. e')\]
above

\[H, f \rightarrow^1 \lambda y^x. e' \rightarrow f \text{ z } H'\]
\[\text{Lemma 15}\]
above

\[f \text{ z } ok\]
above

\[\exists \Gamma, f \rightarrow^1 \lambda y^x. e' = (\lambda x. e) v\]
by substitution

follows by the previous subcase

**subcase** The rest subcases are \(e'_v = f v'\) and \(e'_v = (\lambda x. e') v'\) where \(v'\) is not a variable. Both cases are similar as the previous one, with values stored in the heap first and the expression being \(f z\) as in the first subcase.

**case (match)** match \((C \ v_1 \ldots \ v_n) \ (p_i \rightarrow c_i) \rightarrow e_i[x_1:=v_1, \ldots, x_n:=v_n] \) with \(p_i = C \ x_1 \ldots x_n\).

\[\exists \Gamma, f \rightarrow^1 \lambda y^x. e'\]
above

\[x \mapsto n C \ y_1 \ldots y_n \in H_1\]
follows

\[\exists \Gamma, f H_1 [y_1 = v_1, \ldots, H_1[y_n = v_n]\]
above

\[H_1 \vdash \text{match } x (p_i \rightarrow c_i') \rightarrow, H_1 \mid \text{dup } y's; \text{drop } x; e'_i[xs:=ys]\]
\[(\text{match})\]
given

\[p_i = C \ x\]
above

\[H_1 \vdash \text{match } x (p_i \rightarrow c_i') \rightarrow H_1'\]
given

\[H_1 \vdash x \mapsto H_2\]
\[\text{dmatch}\]
above

\[H_1 = H_2 \implies x\]

\[H_1 \vdash \text{dup } y's; \text{drop } x; e_i[xs:=ys]\]
\[(\text{dup})\]

\[H_1 \vdash y's = H_2 \implies x \mapsto y's\]
\[= H_2 \implies y's \mapsto x\]

\[H_1 \vdash y's \mapsto x = H_2 = y's \mapsto x\]
\[\text{known}\]

\[H_1 \text{ and } H_2 \implies y's\]
\[\text{differs only in } x\]

\[H_2 = y's \implies x = H_2'\]
\[\text{Lemma 15}\]

\[\text{fv}(e_i[xs:=ys]) \in H_2 \implies y's\]
\[\text{Lemma 2}\]

\[H_1 \vdash (e_i[xs:=ys]) \implies H_1 (e_i[xs:=ys]) \implies H_1 (e_i[xs:=ys]) (H_1[e_i[xs:=ys] = (e_i[xs:=ys])\]
\[\text{by substitution}\]

**case (dup)** \(\text{dup } x; e \rightarrow e\)

\[\exists \Gamma, f \rightarrow^1 \lambda y^x. e'\]
above

\[H_1 \vdash \text{dup } x; e' \rightarrow H_1'\]
given

\[x \in H_1\]
follows

\[H_1 \vdash \text{dup } x; e' \rightarrow x \implies e' \rightarrow, H_1 \implies x \implies e' = [H_1] e' = e\]
\[(\text{dup})\] and \(\implies\)

**case (drop)** \(\text{drop } x; e \rightarrow e\)
Lemma 10. (Reference counting semantics is sound (big step, part 1))

Given $E[e_1]$, and $H_1$ ok, and $e'_1$ ok, and $[H_1]e'_1 = E[e_1]$, and $H_1 \vdash e'_1 \vdash H'_1$, then there exists $H_2$, $E''$ and $e'_2$ such that $H_1 \vdash e'_1 \rightarrow^*_{r} H_2 \vdash E'[e'_2]$, and $[H_2]E' = E$, and $[H_2]e'_2 = e_1$.

Proof. (Of Lemma 10) By induction on the evaluation context $E$.

**case** $E = \Box$. Let $E' = \Box$ and $e'_2 = e'_1$, then the goals follow trivially.

**case** $E = E_1 e$.

- $[H_1]e'_1 = E_1[e_1] e$ given
- $e'_1 = e'_2 e'_3$ for some $e'_2, e'_3$
- $[H_1]e'_2 = E_1[e_1]$ follows
- $[H_1]e'_3 = e'$ follows
- $H_1 \vdash e'_2 e'_3 \vdash H'_1$ given
- $H_1 \vdash e'_2 \vdash H_3$ DRAPP
- $H_3 \vdash e'_3 \vdash H'_1$ above
- $H_1 \vdash e'_2 \rightarrow^*_{r} H_2 \vdash E'[e'_2]$ L.H.
- $[H_2]E' = E_1$ above
- $[H_2]e'_2 = e_1$ above
- $E'' = E' e'_3$ let
- $H_1 \vdash e'_2 e'_3 \rightarrow^*_{r} H_2 \vdash E'[e'_2] e'_3$ step
- $H_1 \vdash E'[e'_2] e'_3 \vdash H'_1$ by substitution
- $H_2 \vdash E'[e'_2] e'_3 \vdash H'_1$ Lemma 15
- $fv(e'_2) \in H_2$ Lemma 2
- $[H_2]E' = [H_2]E'[H_2]e'_3$
- $= [H_2]E'[H_1]e'_3$
- $= E_1 e$

**case** $E = x E_1$. 

---

$| H_1 | e'_1 = \text{drop } x; e$ given
$| H_1 | e'_1 = \text{drop } x; e'$ follows
$| H_1 | e' = e$ above
$H_1 \vdash \text{drop } x; e' \vdash H'_1$ given
$H_1 \vdash \text{drop } x; (\cdot) \vdash H_2$ follows
$H_2 \vdash e' \vdash H'_1$ above
$H_1 \vdash H_2 \vdash x$ Lemma 3
$H_1 \vdash \text{drop } x; e' \rightarrow r H_2 \vdash e'$ (drop) and Lemma 4
$H_2 \vdash e' \vdash H'_1$ Lemma 15
$fv(e') \in H_2$ Lemma 2
$| H_2 | e' = [H_1] e' = e$

$\square$
\[ [H_1] e_1' = x \ E_1 [e_1] \]
\[ e_1' = y \ e_3' \]
\[ [H_1] y = x \]
\[ [H_1] e_3' = E_1 [e_1] \]
\[ H_1 \vdash y \ e_3' \vdash H_1' \]
\[ y \in H_3 \]
\[ H_1 \vdash y \vdash H_1' \]
\[ H_1 \vdash H_3 + y \]
\[ H_1 \vdash e_3' \vdash H_1 + y \]
\[ H_1 \vdash e_3' \vdash r, H_2 \ | E'[e_2'] \]
\[ [H_2] E' = E_1 \]
\[ [H_2] e_2' = e_1 \]
\[ E'' = y \ E' \]
\[ H_1 \vdash y \ e_3' \vdash r, H_2 \ | y \ E'[e_2'] \]
\[ H_1 \vdash e_3' \vdash r, H_2 \ | E''[e_2'] \]
\[ y \in H_2 \]
\[ [H_2] E'' = [H_2] y \ [H_2] E' \]
\[ = [H_1] y \ [H_2] E' \]
\[ = x \ E_1 \]

**case** \( E = v \ E_1 \) where \( v \) is not a variable.

\[ [H_1] e_1' = v \ E_1 [e_1] \]
\[ e_1' = v' \ e_3' \]
\[ [H_1] v' = v \]
\[ [H_1] e_3' = E_1 [e_1] \]
\[ H_1 \vdash v' \ e_3' \vdash H_1' \]
\[ H_1 \vdash v' \vdash H_3 \]
\[ H_3 \vdash e_3' \vdash H_1' \]
\[ H_1 \vdash H_3 + fv(v') \]
\[ H_1 \vdash v' \ e_3' \vdash H_1', z \mapsto v' | z \ e_3' \]
\[ H_1, z \mapsto v' \ e_3' \vdash H_1' + fv(v'), z \mapsto v' \]
\[ H_1, z \mapsto v' \ e_3' \vdash r, H_2 \ | E'[e_2'] \]
\[ [H_2] E' = E_1 \]
\[ [H_2] e_2' = e_1 \]
\[ E'' = z \ E' \]
\[ H_1 \vdash v' \ e_3' \vdash r, H_2 \ | z \ E'[e_2'] \]
\[ H_1 \vdash e_3' \vdash r, H_2 \ | E''[e_2'] \]
\[ H_2 \vdash v' \ E'[e_2'] \vdash H_1' \]
\[ fv(v') \in H_2 \]
\[ [H_2] E'' = [H_2] v' \ [H_2] E' \]
\[ = [H_1] v' \ [H_2] E' \]
\[ = v \ E_1 \]

**case** \( E = val x = E_1; e. \)
\[[H_1]_1 e_1' = \text{val } x = E_1[e_1]; e\]
\[e_1' = \text{val } x = e_2'; e_3'\]
\[[H_1]_2 e_2' = E_1[e_1]\]
\[[H_1] e_3' = e\]
\[H_1 \vdash \text{val } x = E_1[e_1]; e \vdash H'_1\]
\[H_1 \vdash e_2' \vdash H_2\]
\[H_1 \vdash e_3' \vdash H_3\]
\[H_2 = E_1\]
\[[H_2] e_2' = e_1\]
\[E'' = \text{val } x = E'; e_3'\]
\[H_1 \vdash \text{val } x = e_2'; e_3' \rightarrow^* H_2 | \text{val } x = E'[e_2']; e_3'\]
\[H_1 \vdash e_3' \rightarrow^* H_2 | E'[e_2']\]
\[H_2 \vdash \text{val } x = E'[e_2']; e_3' \vdash H'_1\]
\[\text{fv}(e_3') \in H_2, [x]\]
\[[H_2] E'' = \text{val } x = [H_2] E'; [H_2] e_3'\]
\[= [H_2] E' [H_1] e_3'\]
\[= E_1 e\]
\[\square\]

\textbf{Lemma 11.} (Reference counting semantics is sound (big step, part 2))

If \(E[e_1] \rightarrow E[e_2]\), and \(H_1 \vdash \text{ok}\), and \(E'[e_3'] \vdash \text{ok}\), and \([H_1] E' = E\), and \([H_1] e_1' = e_1\), and \(H_1 \vdash E[e_1'] \vdash H'_1\), then there exists \(H_2, e_2'\) such that \(H_1 | E'[e_1'] \rightarrow^* H_2 | E'[e_2'], \) and \([H_2] E' = E\), and \([H_2] e_2' = e_2\).

\textbf{Proof.} (Of Lemma 11) By induction on the evaluation context \(E\). Note that following Lemma 15 and 2, we have \(\text{fv}(E') \in H_2\).

- \(H_2 E' = [H_1] E' = E\).
  - \(\text{case } E = \square\). Then \(E' = \square\). The goal follows by Lemma 9.
  - \(\text{case } E = E_1 e\), then \(E' = E_1' e'\).
    \[[H_1] E' = E_1 e\] given
    \[[H_1] E_1' = E'\] above
    \[[H_1] e_1' = e\] above
    \([E_1][e_1] \rightarrow E_1[e_2]\) given
    \(H_1 \vdash E_1'[e_1'] e' \vdash H'_1\) given
    \(H_1 \vdash E_1'[e_1'] \vdash H_2\) \(\text{drapp}\)
    \(H_2 \vdash e' \vdash H'_1\) above
    \(H_1 | E_1'[e_1'] \rightarrow^* H_3 | E_1'[e_2']\) I.H.
    \([H_3] e_2' = e_2\) above
    \(H_1 | E_1'[e_1'] e' \rightarrow^* H_3 | E_1'[e_2'] e'\) \(\rightarrow\)
      - \(\text{case } E = v E_1\), then \(E' = x E_1'\).
    \[[H_1] E' = v E_1\] given
    \[[H_1] x' = v\] above
    \([E_1][e_1] \rightarrow E_1[e_2]\) given
    \(H_1 \vdash E_1'[e_1'] x' \vdash H'_1\) given
    \(H_1 \vdash x \vdash H_2\) \(\text{drapp}\)
    \(H_2 \vdash E_1'[e_1'] \vdash H'_1\) above
    \(H_1 = H_2 + x\) Lemma 3
    \(H_1 \vdash E_1'[e_1'] \vdash H'_1 + x\) Lemma 8
    \(H_1 | E_1'[e_1'] \rightarrow^* H_3 | E_1'[e_2']\) I.H.
    \([H_3] e_2' = e_2\) above
    \(H_1 \vdash x E_1'[e_1'] \rightarrow^* H_3 | x E_1'[e_2']\) \(\rightarrow\)
      - \(\text{case } E = \text{val } x = E_1; e\), then \(E' = \text{val } x = E_1'; e'\).
\([H_1]E' = \text{val } x = E_1; e\) given
\([H_1]E_1' = E'\) above
\([H_1]e' = e\) above
\(E_1[e_1] \mapsto E_1[e_2]\) given
\(H_1 \vdash \text{val } x = E_1'[e_1'] \vdash H_1'\) given
\(H_1 \vdash E_1'[e_1'] + H_2\)
\(H_1 \vdash E_1'[e_1'] \mapsto r, H_3 \vdash E_1'[e_2']\) IH.
\([H_3]e_3' = e_2\) above
\(H_1 \vdash \text{val } x = E_1'[e_1'] \mapsto r, H_3 \vdash \text{val } x = E_1'[e_2']\)
\(\square\)

Lemma 12. (Reference counting semantics is sound (big step))
If \(e_1 \mapsto e_2\) and \(H_1 \vdash \text{ok}\) and \(e_1' \vdash \text{ok}\) and \([H_1]e_1' = e_1\) and \(H_1 \vdash e_1' + H_1'\) then there exists \(H_2\) and \(e_2'\) such that \(H_1 \vdash e_1' \mapsto r, H_2 \vdash e_2',\) and \([H_2]e_2' = e_2\).

Proof. (Of Lemma 12)
\(e_1 \mapsto e_2\) given
\(e_1 = E[e_3]\) suppose
\(e_2 = E[e_4]\) suppose
\(E[e_3] \mapsto E[e_4]\) given
\(e_1' = E_1[e_1']\) suppose
\(H_1 \vdash E_1'[e_2'] \mapsto r, H_2 \vdash E_2[e_3']\) Lemma 10
\([H_2]E_2 = E\) above
\([H_2]e_3' = e_3\) above
\(H_2 \vdash \text{ok}\) Lemma 15
\(E_2[e_3'] \vdash \text{ok}\) above
\(H_2 \vdash E_2[e_3'] + H_1'\) above
\(H_2 \vdash E_2[e_3'] \mapsto H_3 \vdash E_2[e_4']\) Lemma 11
\([H_3]E_2 = E\) above
\([H_3]e_4' = e_4\) above
\([H_3](E_2[e_4']) = ([H_3]E_2)[[H_3]e_4']\) by substitution
\(= E[e_4] = e_2\)
\(\square\)

Proof. (Of Theorem 1)
\(\emptyset \vdash \emptyset + e \rightarrow e'\) given
\(e \rightarrow^* v\) given
\(e' \rightarrow^* v\) Theorem 5
\(H \vdash \text{ok}\)
\(e' \vdash \emptyset\) from LAM
\(\emptyset \vdash e' \rightarrow \emptyset\) Lemma 7
\(e' \rightarrow^* r, H_2 \vdash v'\) Lemma 12
\([H_2]v' = v\) above
\(\text{case } v' = x.\) Then the goal is proved.
\(\text{case } v'\) is not a variable. Then by (lam) or (con) we have \(H_2 \vdash v' \rightarrow^1 r, H_2, z \mapsto^1 v' | z,\) with \(z\) fresh. Then \([H_2, z \mapsto^1 v']z = [H_2, z \mapsto^1 v']v' = [H_2]v' = v.\) \(\square\)

D.3 No Garbage
D.3.1 Extending strict evaluation semantics. If we add \texttt{dup } e\ and \texttt{drop } e\ in the syntax, as well as add to the standard semantics in Figure 6 the following rules:
\((\texttt{dup})\) \(\texttt{dup } e'; e \rightarrow e\)
\((\texttt{drop})\) \(\texttt{drop } e'; e \rightarrow e\)
we immediately see that translations does not change evaluation:

**Theorem 5.** (Translation is sound)
If \( e \leftrightarrow^* v \) with \( \emptyset \vdash \emptyset \vdash e \leadsto e' \), then also \( e' \leftrightarrow^* v \).

**Proof.** (Of Theorem 5) Follows directly from Lemma 13 and the two reduction rule \((\text{dup})\) and \((\text{drop})\). \(\square\)

**Lemma 13.** (Translation only inserts \(\text{dup/drop}\))
If \( \Delta \vdash e \leadsto e' \) then \( e = [e'] \).

**Proof.** (Of Lemma 13) By straightforward case analysis of each derivation. \(\square\)

D.3.2 Evaluation retains Heap Reference Counting.

**Definition 4.** (Well-formed Abstractions)
If \( e \) ok, then all \((\lambda^{ys} x. e_1)\) in \( e \) satisfies \( [ys, x] \vdash e_1 \vdash \emptyset \).

**Definition 5.** (Well-formed Heap)
If \( H \) ok, then (1) if \( x \mapsto^1 v \in H \), then \( fv(v) \in H \), and \( v \) ok; (2) there is no dependency cycles in \( H \).

**Lemma 14.** (No Garbage (Small step))
Given \( H_1\) ok and \( e_1 \) ok if \( H_1 \vdash e_1 \vdash H' \), and \( H_1 \vdash e_1 \rightarrow_r H_2 \vdash e_2 \), then \( H_2 \) ok, \( e_2 \) ok, and \( H_2 \vdash e_2 \vdash H' \).

**Proof.** (Of Lemma 14) When we a new variable \( z \mapsto^1 v \) in the heap (e.g., \((\text{lam})\)), \( z \) is fresh so its domain cannot refer to \( z \) (even indirectly). So there is no dependency cycle. Also, in those cases, since \( H_1 \vdash v \vdash H' \), by Lemma 2, we know \( fv(v) \in H_1 \). Moreover we have \( v \) ok as a preconditions.

Heap reduction retains abstractions, with the only change being substitution. If \( \langle [ys, x] \vdash e \vdash \emptyset \rangle \), then \( \langle [ys[y:=z], x] \vdash e[y:=z] \vdash \emptyset \rangle \) by substitution.

Now we prove \( H_2 \vdash e_2 \vdash H' \) by induction on the judgment.

**case** \((\text{app})\) \( H \mid f \ x \rightarrow_r H \mid \ \text{dup} \ ys; \ \text{drop} \ f; \ e \ [x:=z] \ (f \mapsto^n \lambda^{ys} x. e) \in H \)

\[ H \vdash f \vdash H_1 \]
\[ H = H_1 + f + z \]
\[ ys \in H_1 + f + z \]
\[ H \vdash \text{dup} \ ys; \ (\cdot) \vdash H + ys \]
\[ H + ys = H_1 + f + z + ys \]
\[ = H_1 + ys + z + f \]
\[ \text{Lemma 5} \]
\[ H_1 + ys + z + f + \ \text{drop} \ f; \ (\cdot) \vdash H_1 + ys + z \]
\[ \langle [ys, x] \vdash e \vdash \emptyset \rangle \]
\[ \lambda^{ys} x. e \]
\[ \text{ok} \]
\[ \langle [ys, z] \vdash e \ [x:=z] + \emptyset \rangle \]
\[ \text{by substitution} \]
\[ H_1 + ys + z + e \ [x:=z] + H_1 \]
\[ \text{Lemma 8} \]

**case** \((\text{match})\) \( H \mid \text{match} \ x \ P_i \rightarrow e_i \rightarrow_r H \mid \text{dup} \ ys; \ \text{drop} \ x; \ e_i[x:=ys] \) \( i = 1, \ldots, n \) \( C \) \( x \rightarrow^n C \) \( ys \) \( \in H \)

\[ H \vdash \text{match} \ x \ P_i \rightarrow e_i \vdash H' \]
\[ \text{given} \]
\[ H \vdash x \vdash H_1 \]
\[ H_1, [x] \vdash e_i \vdash H' \]
\[ \text{given} \]
\[ xs \notin H_i \]
\[ \text{given} \]
\[ H = H_1 + x \]
\[ \text{Lemma 3} \]
\[ (x \rightarrow^n C) \ ys \in H \]
\[ \text{given} \]
\[ ys \in H \]
\[ H \text{ ok} \]
\[ H \vdash \text{dup} \ ys; \ (\cdot) \vdash H + ys \]
\[ \text{by definition} \]
\[ H + ys = H_1 + x + ys \]
\[ = H_1 + ys + x \]
\[ \text{Lemma 5} \]
\[ H_1 + ys + x \vdash \text{drop} \ x; \ (\cdot) \vdash H + ys \]
\[ \text{Lemma 4} \]
\[ H_1 + ys + e_i[ys:=ys] \vdash H' \]
\[ \text{by substitution} \]

**case** \((\text{lam})\) \( H \mid (\lambda^{ys} x. e) \rightarrow^h H, f \mapsto^1 \lambda^{ys} x. e \)

\[ H \text{ fresh } f \]
The ok part reasoning of Lemma 14 can be easily generalized to big step. So from now on we will implicitly assume every expression and heap we discuss is ok.

**Lemma 15.** (No Garbage (big step))

If \( H_1 \vdash E[e_1] \vdash H' \), and \( H_1 \vdash E[e_1] \longrightarrow_r H_2 \vdash E[e_2] \), then \( H_2 \vdash E[e_2] \vdash H' \).

**Proof.** (Proof for Lemma 15) By induction on \( E \).

case \( E = \square \). Follows by Lemma 14.

case \( E = E_1 \ e \).

\( H_1 \vdash E_1[e_1] \vdash H_2 \) given

\( H_1 \vdash E_1[e_1] \vdash H_3 \) drapp

\( H_3 \vdash e \vdash H_2 \) given

\( H_1 \vdash E_1[e_2] \vdash H_3 \) IH.

\( H_1 \vdash E_1[e_2] \vdash H_2 \) drapp

case \( E = x \ E_1 \).

\( H_1 \vdash x \ E_1[e_1] \vdash H_2 \) given

\( H_1 \vdash x \ E_1[e_1] \vdash H_3 \) drapp

\( H_3 \vdash E_1[e_1] \vdash H_2 \) given

\( H_3 \vdash E_1[e_2] \vdash H_2 \) IH.

\( H_1 \vdash x \ E_1[e_2] \vdash H_2 \) drapp

case \( E = \text{val } x = E_1[e_1]; e \).

\( H_1 \vdash \text{val } x = E_1[e_1]; e \vdash H_2 \) given

\( H_1 \vdash E_1[e_1] \vdash H_3 \) dbind

\( H_3, x \vdash^1 () \vdash e \vdash H_2 \) given

\( x \notin H_2 \) given

\( H_1 \vdash E_1[e_2] \vdash H_3 \) IH.

\( H_1 \vdash \text{val } x = E_1[e_2]; e \vdash H_2 \) dbind

case \( E = C \ x_1 \ldots x_n \ E_1 \ v_j \ldots v_n \).
H₁ ⊢ C x₁ ... xᵢ E₁[e₁] vⱼ ... vₙ ⊢ H₂  given
H₁ ⊢ x₁ ... xᵢ ⊢ H₃  DRCON
H₃ ⊢ E₁[e₁] ⊢ H₄  given
H₄ ⊢ vⱼ ... vₙ ⊢ H₂  given
H₃ ⊢ E₁[e₂] ⊢ H₄  I.H.
H₁ ⊢ C x₁ ... xᵢ E₁[e₂] vⱼ ... vₙ ⊢ H₂  DRCON

Theorem 6. (No garbage)
Given ∅; ∅ ⊢ e ⊢ e₁, and ∅ | e₁ ⊢ r, H₁ | eᵢ, then Hᵢ ⊢ eᵢ ⊢ ∅.

Proof. (Of Theorem 6)

∅ ok  by construction
e₁ ok  by LAM
∅ ⊢ e₁ ⊢ ∅  Lemma 7
Hᵢ ⊢ eᵢ ⊢ ∅  Lemma 15

D.3.3 No Garbage.

Lemma 16. (Reachability)
If H₁ ⊢ e ⊢ H₁, then there for all y ∈ dom(H₁) − dom(H₂), reach(y, H₁ | e). For ease of reference, we denote it as reach(H₁ − H₂, H₁ | e).

Proof. (Of Lemma 16) By induction on the judgment.
case
H₀ ⊢ C v₁ ... vₙ ⊢ H_n  given
H₀ ⊢ v₁ ⊢ H₁ ... Hₙ₋₁ ⊢ vₙ ⊢ H_n  DRCON
reach(Hᵢ₋₁ − Hᵢ, Hᵢ₋₁ | vᵢ)  I.H.
reach(Hᵢ₋₁ − Hᵢ, H₀ | vᵢ)  Lemma 2
reach(Hₙ − H₀, H₀ | C v₁ ... vₙ)  Follows
case
H, x ⊢ x ⊢ x ⊢ H, x ⊢ v  given
dom(H, x ⊢ v) − dom(H, x ⊢ v) = ∅
case
H, x ⊢ λ ys z. e ⊢ x ⊢ H₁  given
H ⊢ ys ⊢ H₁  DRVARLAM
reach(H − H₁, H | ys)  I.H.
reach(H − H₁, H | λ ys z. e)  by definition
reach((H, x ⊢ λ ys z. e) − H₁, (H, x ⊢ λ ys z. e) | x)  follows
case
H, x ⊢ C ys ⊢ x ⊢ H₁  given
H ⊢ ys ⊢ H₁  DRVARCON
reach(H − H₁, H | ys)  I.H.
reach(H − H₁, H | C ys)  by definition
reach((H, x ⊢ C ys) − H₁, (H, x ⊢ C ys) | x)  follows
case
\[ H, x \mapsto^n v \vdash \text{dup } x; e \vdash H_1 \] given
\[ H, x \mapsto^{n+1} v \vdash e \vdash H_1 \] \text{dirdup}
\[ \text{reach}(H, x \mapsto^{n+1} v) - H_1, (H, x \mapsto^{n+1} v) \mid e) \] I.H.
\[ \text{reach}(H, x \mapsto^{n} v) - H_1, (H, x \mapsto^{n} v) \mid \text{dup } x; e) \] follows
\textbf{case}

\[ H, x \mapsto^{n+1} v \vdash \text{drop } x; e \vdash H_1 \] given
\[ H, x \mapsto^{n} v \vdash e \vdash H_1 \] \text{dirdrop}
\[ \text{reach}(H, x \mapsto^{n+1} v) - H_1, (H, x \mapsto^{n+1} v) \mid e) \] I.H.
\[ \text{reach}(H, x \mapsto^{n} v) - H_1, (H, x \mapsto^{n} v) \mid \text{drop } x; e) \] follows
\textbf{case}

\[ H, x \mapsto^{1} C \ y s \vdash \text{drop } x; e \vdash H_1 \] given
\[ H \vdash \text{drop } y s; e \vdash H_1 \] \text{dirdropcon}
\[ \text{reach}(H - H_1, H \mid \text{drop } y s; e) \] I.H.
\[ \text{reach}(H, x \mapsto^{1} C \ y s) - H_1, (H, x \mapsto^{1} C \ y s) \mid \text{drop } x; e) \] follows
\textbf{case}

\[ H, x \mapsto^{1} \lambda y z e \vdash \text{drop } x; e \vdash H_1 \] given
\[ H \vdash \text{drop } y s; e \vdash H_1 \] \text{dirdroplam}
\[ \text{reach}(H - H_1, H \mid \text{drop } y s; e) \] I.H.
\[ \text{reach}(H, x \mapsto^{1} \lambda y z e) - H_1, (H, x \mapsto^{1} \lambda y z e) \mid \text{drop } x; e) \] follows
\textbf{case}

\[ H \vdash \lambda y \ y s \ e \vdash H_1 \] given
\[ H \vdash y s \vdash H_1 \] \text{drlam}
\[ \text{reach}(H - H_1, H \mid y s) \] I.H.
\[ \text{reach}(H - H_1, H \mid \lambda y \ y s \ e) \] follows
\textbf{case}

\[ H \vdash e_1 e_2 \vdash H_2 \] given
\[ H \vdash e_1 \vdash H_1 \] \text{drapp}
\[ H_1 \vdash e_2 \vdash H_2 \] \text{drapp}
\[ \text{reach}(H - H_1, H \mid e_1) \] I.H.
\[ \text{reach}(H_1 - H_2, H_1 \mid e_2) \] I.H.
\[ \text{reach}(H_1 - H_2, H \mid e_2) \] Lemma 2
\[ \text{reach}(H - H_2, H \mid e_1 e_2) \] follows
\textbf{case}

\[ H \vdash \text{val } x = e_1 ; e_2 \vdash H_2 \] given
\[ H \vdash e_1 \vdash H_1 \] \text{drbind}
\[ H_1, x \mapsto^{1} () \vdash e_2 \vdash H_2 \] \text{drbind}
\[ \text{reach}(H - H_1, H \mid e_1) \] I.H.
\[ \text{reach}((H_1, x \mapsto^{1} ()) - H_2, (H_1, x \mapsto^{1} ()) \mid e_2) \] I.H.
\[ \text{reach}((H_1, x \mapsto^{1} ()) - H_2, (H, x \mapsto^{1} ()) \mid e_2) \] Lemma 2
\[ \text{dom}(H_1) \subseteq \text{dom}(H_1, x \mapsto^{1} ()) \]
\[ \text{reach}(H_1 - H_2, (H, x \mapsto^{1} ()) \mid e_2) \] follows
\[ x \notin H \] known
\[ x \notin \text{dom}(H) - \text{dom}(H_2) \] follows
\[ \text{reach}(H - H_2, H \mid \text{val } x = e_1 ; e_2) \] follows
\textbf{case}
\[ \begin{align*}
\text{H} \vdash \text{match } x \{ \bar{p}_i \mapsto e_i \} & \implies \text{H}' & \text{given} \\
\text{H} \vdash x \vdash H_1 & \text{DRMATCH} \\
H_1, [\text{bv}(p_i)] \vdash e_i \vdash H' & \text{DRMATCH} \\
\text{bv}(p_i) \notin H, H' & \text{DRMATCH} \\
\text{reach}(H - H_1, H \mid x) & \text{I.H.} \\
\text{reach}((H_1, [\text{bv}(p_i)]) - H', (H_1, [\text{bv}(p_i)]) \mid e_i) & \text{I.H.} \\
\text{reach}((H_1, [\text{bv}(p_i)]) - H', (H, [\text{bv}(p_i)]) \mid e_i) & \text{Lemma 2} \\
\text{dom}(H_1) \subseteq \text{dom}(H, [\text{bv}(p_i)]) & \text{follows} \\
\text{bv}(p_i) \notin H & \text{known} \\
\text{bv}(p_i) \notin \text{dom}(H - \text{dom}(H')) & \text{follows} \\
\text{reach}(H - H', H \mid \text{match } x \{ \bar{p}_i \mapsto e_i \}) & \text{follows}
\end{align*} \]

\[ \square \]

**Proof. (Of Theorem 2)**

\[ \begin{align*}
\varnothing; \varnothing \vdash e \rightsquigarrow e' & \text{given} \\
H_i \vdash e_i \vdash \varnothing & \text{Theorem 6} \\
\text{reach}(H_i - \varnothing, H_i \mid e_i) & \text{Lemma 16} \\
\square
\end{align*} \]

**D.4 Soundness of Syntax-directed Translation**

**Proof. (Of Theorem 3)** By induction on the judgment.

case

\[ \begin{align*}
\Delta \mid x \Downarrow x \rightsquigarrow x & \text{ given} \\
\Delta \mid x \vdash x \rightsquigarrow \text{VAR} \\
\end{align*} \]

case

\[ \begin{align*}
\Delta, x \mid \varnothing \Downarrow x \rightsquigarrow \text{dup } x; x & \text{ given} \\
\Delta, x \mid x \vdash x \rightsquigarrow x & \text{VAR} \\
\Delta, x \mid \varnothing \vdash x \rightsquigarrow \text{dup } x; x & \text{DUP} \\
\end{align*} \]

case

\[ \begin{align*}
\Delta, \Delta_1 \mid \Gamma \Downarrow x \rightsquigarrow \text{d}up \Delta_1; \lambda^{\gamma x} x. e' & \text{ given} \\
x \in \text{fv}(e) & \text{SLAM} \\
\varnothing \mid ys, x \Downarrow x \rightsquigarrow e' & \text{SLAM} \\
ys = \text{fv}(\lambda x. e) & \text{SLAM} \\
\Delta_1 = ys - \Gamma & \text{SLAM} \\
\varnothing \mid ys, x \vdash e \rightsquigarrow e' & \text{I.H.} \\
\Delta, \Delta_1 \mid \Gamma, \Delta_1 \Downarrow x \rightsquigarrow \lambda^{\gamma x} x. e' & \text{LAM} \\
\Delta, \Delta_1 \mid \Gamma \Downarrow x \rightsquigarrow \text{dup } \Delta_1; \lambda^{\gamma x} x. e' & \text{DUP} \\
\end{align*} \]

case

\[ \begin{align*}
\Delta, \Delta_1 \mid \Gamma \Downarrow x \rightsquigarrow \text{d}up \Delta_1; \lambda^{\gamma x} x. (\text{drop } x; e') & \text{ given} \\
x \notin \text{fv}(e) & \text{SLAM-DROP} \\
\varnothing \mid ys \Downarrow x \rightsquigarrow e' & \text{SLAM-DROP} \\
ys = \text{fv}(\lambda x. e) & \text{SLAM-DROP} \\
\Delta_1 = ys - \Gamma & \text{SLAM-DROP} \\
\varnothing \mid ys, x \vdash e \rightsquigarrow e' & \text{I.H.} \\
\varnothing \mid ys, x \vdash e \rightsquigarrow \text{drop } x; e' & \text{DROP} \\
ys = \text{fv}(\lambda x. \text{drop } x; e) & \text{follows} \\
\Delta, \Delta_1 \mid \Gamma, \Delta_1 \Downarrow x \rightsquigarrow \lambda^{\gamma x} x. \text{drop } x; e' & \text{LAM} \\
\Delta, \Delta_1 \mid \Gamma \Downarrow x \rightsquigarrow \text{d}up \Delta_1; \lambda^{\gamma x} x. e' & \text{DUP}
\end{align*} \]
D.5 Precision

D.5.1 A Heap Reference Counting Calculus for the Algorithm.
\[
\begin{align*}
\text{H} \vdash s \, v_1 + H_1 \ldots H_{n-1} \vdash s \, v_n + H_n & \quad \text{[SRCON]} \\
\text{H} \vdash s \, C \, v_1 \ldots v_n + H_1 & \quad \text{[SRVAR]} \\
\text{H}, x \mapsto^{n+1} v \vdash s \, x + H, x \mapsto^n v & \quad \text{[SRVARLAM]} \\
\text{H} \vdash s \, \lambda^{ys} \, x \, . \, e \vdash s \, x + H_1 & \quad \text{[SRVARCON]} \\
\text{H} \vdash s \, ys + H_1 & \quad \text{[SRLAM1]} \\
\text{H} \vdash s \, ys + H_1 & \quad \text{[SRLAM2]} \\
\text{H} \vdash s \, e_1 + H_1 & \quad \text{[SRAPP]} \\
\text{H} \vdash s \, e_1 + H_1, x \mapsto^1 () \vdash s \, e_2 + H_2 & \quad \text{[SRBIND1]} \\
\text{H} \vdash s \, e_1 + H_1, x \mapsto^1 () \vdash s \, e_2 + H_2 & \quad \text{[SRBIND2]} \\
\text{H} \vdash s \, x + H_1 & \quad \text{[SRMATCH]} \\
\text{H} \vdash s \, e + H_1 & \quad \text{[SRDUPLAM]} \\
\text{H}, x \mapsto^n v \vdash s \, e + H_1 & \quad \text{[SRDUP]} \end{align*}
\]

Where drop, and dup followed by drop are only allowed in:

\[
\begin{align*}
\text{H}, x \mapsto^{n+1} v \vdash s \, e + H_1 & \quad \text{[RRDUP]} \\
\text{H}, x \mapsto^n v \vdash s \, \text{dup} \, x; e + H_1 & \quad \text{[RRDUAL]} \\
\text{H}, x \mapsto^n v \vdash s \, e + H_1 & \quad \text{[RRDROP]} \\
\text{H} \vdash s \, \text{drop} \, x; e + H_1 & \quad \text{[RRDROPUNIT]} \\
\text{H}, x \mapsto^1 () \vdash s \, \text{drop} \, x; e + H_1 & \quad \text{[RRDROPLAM]} \\
\text{H}, x \mapsto^1 C \, ys \vdash s \, \text{drop} \, x; e + H_1 & \quad \text{[RRDROPCONS]} \\
\end{align*}
\]
$$
\frac{H \vdash s \ e \ + \ H_1}{H \vdash r \ e \ + \ H_1} \ [\text{rrsub}]
$$

D.5.2 Properties. Those properties are essentially the same as the lemma for the heap reference counting calculus.

Definition 6. (Well-formed Abstractions)
If $e \in E$, then all $(\lambda y \ x. e_1)$ in $e$ satisfies (1) $[ys, \ x] \vdash e_1 + \emptyset$; or (2) $e_1 = \text{drop} \ x$; $e_2$, and $[ys] \vdash e_2 + \emptyset$.

Definition 7. (Well-formed Heap)
If $H \in H$, then (1) if $x \mapsto^n v \in H$, then $fv(v) \in H$, and $v \in E$; (2) there is no dependency cycles in $H$.

Lemma 17. (Heap Reference Counting Free variables)
If $H_1 \vdash r \ e \ + \ H_2$ or $H_1 \vdash s \ e \ + \ H_2$, then $fv(e) \in H_1$, and $fv(H_2) \in H_1$ with same domains.

Lemma 18. (Drop is dual to extension)
If $H_1 \vdash r \ \text{drop} \ x$; $() + H_2$, then $H_1 = H_2 + x$. Similarly, if $H_1 \vdash s \ x + H_2$ or $H_1 \vdash s \ x + H_2$, then $H_1 = H_2 + x$.

Lemma 19. (Extension is dual to drop)
$H + x \vdash r \ \text{drop} \ x$; $() + H$. Similarly, $H + x \vdash s \ x + H$ and $H + x \vdash s \ x + H$.

Lemma 20.
$\Gamma, x \vdash s \ x + \ G$. Similarly, if $[\Delta] \vdash r \ e + H$, then $[\Gamma, x] \vdash r \ \text{drop} \ x$; $e + H$.

Theorem 7. (No garbage)
Given $\emptyset, \emptyset \vdash r \ e \mapsto e_1$, and $\emptyset | e_1 \mapsto^* r H_i | e_i$, then $H_i \vdash r e_1 + \emptyset$.

D.5.3 Relating to linear resource calculus.

Lemma 21. (Algorithmic linear resource calculus relates to reference counting)
If $\Delta | \Gamma \vdash s \ e \mapsto e'$, then $[\Delta, \Gamma] \vdash s e' + [\Delta]$. By rrsub we also have $[\Delta, \Gamma] \vdash r e' + [\Delta]$.

Proof. (Of Lemma 21) During the proof, we reply on the fact that source program (i.e., $e$) has no drop or dup. By induction on the elaboration.

case

$\Delta | x \vdash s \ x \mapsto x$ \hspace{1cm} given

$[\Delta, x] \vdash s \ x + [\Delta]$ \hspace{1cm} Lemma 20

case

$\Delta, x | \emptyset \vdash s \ x \mapsto \text{dup} \ x; x$ \hspace{1cm} given

$[\Delta, x] + x + s \ x + [\Delta, x]$ \hspace{1cm} Lemma 19

$[\Delta, x] \vdash s \ \text{dup} \ x; x + [\Delta, x]$ \hspace{1cm} Srdup

case

$\Delta, \Delta_1 | \Gamma \vdash s \lambda x. e \mapsto \text{dup} \Delta_1; \lambda y s \ x. e'$ \hspace{1cm} given

$x \in \text{fv}(e)$ \hspace{1cm} Slam

$\emptyset | y s, x \vdash s \ e' \mapsto e'$ \hspace{1cm} Slam

$y s \equiv \text{fv}(\lambda x. e)$ \hspace{1cm} Slam

$\Delta_1 = y s - \Gamma$ \hspace{1cm} Lh.

$[\Delta, \Delta_1, y s] \vdash s y s + [\Delta, \Delta_1]$ \hspace{1cm} Lemma 19

$[\Delta, \Delta_1, \Gamma, \Delta_1] \vdash s y s + [\Delta, \Delta_1]$ \hspace{1cm} by substitution

$[\Delta, \Delta_1, \Gamma, \Delta_1] \vdash s \lambda y s x. e' + [\Delta, \Delta_1]$ \hspace{1cm} Srdam

$[\Delta, \Delta_1, \Gamma] \vdash s \ \text{dup} \Delta_1; \lambda y s x. e' + [\Delta, \Delta_1]$ \hspace{1cm} Srdup
\[\Delta, \Delta_1 \mid \Gamma \vdash \lambda x. e \leadsto \text{dup } \Delta_1; \lambda^y x. (\text{drop } x; e') \quad \text{given}\]
\[x \notin \text{fv}(e)\]
\[\emptyset \mid y s \Gamma \vdash e \leadsto e'\]
\[y s \equiv \text{fv}(\lambda x. e)\]
\[\Delta_1 = y s - \Gamma\]
\[\[y s\] \vdash_s e' \implies \emptyset\]
\[\[\Delta, \Delta_1, y s\] \vdash_s y s \equiv \text{I.H.} \quad \text{Lemma 19}\]
\[\[\Delta, \Delta_1, \Gamma\] \vdash_s y s \equiv \text{by substitution}\]
\[\[\Delta, \Delta_1, \Gamma\] \vdash_s \lambda^y x. \text{drop } x; e \equiv \text{SRLAM2}\]
\[\[\Delta, \Delta_1, \Gamma\] \vdash_s \text{dup } \Delta_1; \lambda^y x. \text{drop } x; e' \equiv \text{SRDUP}\]

\[\Delta \mid \Gamma \vdash e_1 e_2 \equiv e'_1 e'_2 \quad \text{given}\]
\[\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash e_1 \equiv e'_1 \quad \text{SAPP}\]
\[\Delta \mid \Gamma_2 \vdash e_2 \equiv e'_2 \quad \text{SAPP}\]
\[\Gamma_2 = \Gamma \cap \text{fv}(e_2) \quad \text{SAPP}\]
\[\[\Delta, \Gamma\] \vdash_s e'_1 \equiv \text{I.H.}\]
\[\[\Delta, \Gamma\] \vdash_s e'_2 \equiv \text{I.H.}\]
\[\[\Delta, \Gamma\] \vdash_s \equiv \text{SRAPP}\]

\[\Delta \mid \Gamma \vdash \text{val } x = e_1; e_2 \equiv \text{val } x = e'_1; e'_2 \quad \text{given}\]
\[x \in \text{fv}(e_2)\]
\[x \notin \Delta, \Gamma\]
\[\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash e_1 \equiv e'_1 \quad \text{SBIND}\]
\[\Delta \mid \Gamma_2 \vdash e_2 \equiv e'_2 \quad \text{SBIND}\]
\[\Gamma_2 = \Gamma \cap (\text{fv}(e_2) - x) \quad \text{SBIND}\]
\[\[\Delta, \Gamma\] \vdash_s e'_1 \equiv \text{I.H.}\]
\[\[\Delta, \Gamma\] \vdash_s e'_2 \equiv \text{I.H.}\]
\[\[\Delta, \Gamma\] \vdash_s \equiv \text{SRBIND1}\]

\[\Delta \mid \Gamma \vdash \text{val } x = e_1; e_2 \equiv \text{val } x = e'_1; e'_2 \quad \text{given}\]
\[x \notin \text{fv}(e_2), \Delta, \Gamma\]
\[\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash e_1 \equiv e'_1 \quad \text{SBIND-DROP}\]
\[\Delta \mid \Gamma_2 \vdash e_2 \equiv e'_2 \quad \text{SBIND-DROP}\]
\[\Gamma_2 = \Gamma \cap \text{fv}(e_2) \quad \text{SBIND-DROP}\]
\[\[\Delta, \Gamma\] \vdash_s e'_1 \equiv \text{I.H.}\]
\[\[\Delta, \Gamma\] \vdash_s e'_2 \equiv \text{I.H.}\]
\[\[\Delta, \Gamma\] \vdash_s \equiv \text{SRBIND2}\]
D.5.4 Precision.

**Lemma 22.** (Reachability for erased expressions)
If $H_1 \vdash_{e} e \rightarrow H_2$, then reach($H_1 - H_2$, $H_1 \mid [e]$).

**Proof.** (Of Lemma 22) By induction on the judgment.

**case**

$H \vdash_{s} C v_1 \ldots v_n \rightarrow C v'_1 \ldots v'_n$

$\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \vdash_{s} v_i \rightarrow v'_i$

Assume $\text{bv}(p_i)$ fresh

By definition of $\text{fv}$

By induction on the judgment.

**case**

$H, x \rightarrow^{n+1} v \vdash_{s} x \rightarrow^{n} v$

$\text{dom}(H, x \rightarrow^{n+1} v) - \text{dom}(H, x \rightarrow^{n} v) = \varnothing$

$H, x \rightarrow^{1} \lambda y z. e \vdash_{s} x \rightarrow H_1$

$H \vdash_{s} y s \rightarrow H_1$

$\text{reach}(H - H_1, H \mid [y s])$

By definition

$\text{reach}(H, x \rightarrow^{1} \lambda y z. e) \rightarrow H_1$, $(H, x \rightarrow^{1} \lambda y z. e) \mid [x])$

by definition
H, x ↦⁺¹ C ys ⊢ x − | H₁
H ⊢ s ys ⊢ H₁
reach(H − H₁, H | [ys]) LH.
reach(H − H₁, H | [C ys]) by definition
reach((H, x ↦⁺¹ C ys) − H₁, (H, x ↦⁺¹ C ys) | [x]) follows case

H ⊢ s λ²⁰ x. e ⊢ H₁ given
H ⊢ s λ²⁰ x. e ⊢ H₁ srlam1
reach(H − H₁, H | [λ²⁰ x. e]) LH.
reach(H − H₁, H | [λ²⁰ x. e]) follows case

H ⊢ s e₁ e₂ ⊢ H₂ given
H ⊢ s e₁ ⊢ H₁ srapp
H₁ ⊢ s e₂ ⊢ H₂ srapp
reach(H − H₁, H | [e₁]) LH.
reach(H₁ − H₂, H₁ | [e₂]) LH.
reach(H₁ − H₂, H | [e₂]) Lemma 17
reach(H − H₂, H | [e₁ e₂]) follows case

H ⊢ s val x = e₁ ; e₂ ⊢ H₂ given
H ⊢ s e₁ ⊢ H₁ srbind1
H₁, x ↦⁺¹ () ⊢ s e₂ ⊢ H₂ srbind1
x ∉ H, H₂ srbind1
reach(H − H₁, H | [e₁]) LH.
reach((H₁, x ↦⁺¹ ()) − H₂, (H₁, x ↦⁺¹ ()) | [e₂]) LH.
reach(((H₁, x ↦⁺¹ ()) − H₂, (H, x ↦⁺¹ ()) | [e₂]) Lemma 17
dom(H₁) ⊆ dom(H₁, x ↦⁺¹ ())
reach(H₁ − H₂, (H, x ↦⁺¹ ()) | [e₂]) follows known
x ∉ H
x ∉ dom(H) − dom(H₂) follows
reach(H − H₂, H | [val x = e₁ ; e₂]) follows case

H ⊢ s val x = e₁ ; drop x; e₂ ⊢ H₂ given
H ⊢ s e₁ ⊢ H₁ srbind2
H₁ ⊢ s e₂ ⊢ H₂ srbind2
x ∉ H srbind2
reach(H − H₁, H | [e₁]) LH.
reach((H₁ − H₂, H₁ | [e₂]) LH.
reach((H₁ − H₂, H | [e₂]) Lemma 17
x ∉ H known
x ∉ dom(H) − dom(H₂) follows
reach(H − H₂, H | [val x = e₁ ; drop x; e₂]) follows case
H ⊢ s match x { pᵢ ↦→ drop ysᵢ; eᵢ } ⊣ H'

H ⊢ s x ∈ H₁

H₁, [[bv(pᵢ)]] ⊢ r drop ysᵢ; () ⊣ Hᵢ

Hᵢ ⊢ s eᵢ ⊣ H'

bv(pᵢ) ∉ H, H'

ysᵢ ⊆ fv(⌈eᵢ⌉) ∪ bv(pᵢ)

reach(H – H₁, H | x)  

reach(H₁, [[bv(pᵢ)]] = Hᵢ + ysᵢ  

reach((Hᵢ – H', Hᵢ | [eᵢ])) 

reach((Hᵢ – H', (H₁,[[bv(pᵢ)]] | [eᵢ]))  

ysᵢ ⊆ fv(⌈eᵢ⌉) ∪ bv(pᵢ)

reach(Hᵢ + ysᵢ – H', H | bv(pᵢ); [eᵢ])  

follows

reach(Hᵢ,[[bv(pᵢ)]] – H', H | bv(pᵢ); [eᵢ])

by substitution

bv(pᵢ) ∉ H

bv(pᵢ) ∉ dom(H) – dom(H')

follows

reach(H – H', H | match x { pᵢ ↦→ [eᵢ] })

follows

case

H, x ↦ n v ⊢ s dup x; e ⊣ H₁

H, x ↦ n+1 v ⊢ s e ⊣ H

x ∈ fv(⌈e⌉)

reach((H, x ↦ n v) – H, (H, x ↦ n+1 v) | e)  

L.H.

reach((H, x ↦ n v) – H, (H, x ↦ n v) | e)  

Follows

□

Proof. (Of Theorem 4)

∅ | ∅ r e ⇝ e'

given

∅ ⊢ r e' ⇝ ∅  

Lemma 21

Hᵢ ⊢ r eᵢ ⇝ ∅  

Theorem 7

eᵢ ≠ E[ dup x; eᵢ]

given

Hᵢ ⊢ s e' ⇝ ∅  

rrsub

reach(Hᵢ – ∅, Hᵢ | [e])  

Lemma 22

all y ∈ dom(Hᵢ), reach(y, Hᵢ | eᵢ)

follows

□