Perceus: Garbage Free Reference Counting with Reuse

Microsoft Technical Report, MSR-TR-2020-42, Jan 11, 2021, v3.

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 run-time. 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, λ^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 λ^1, and prove

*The first two authors contributed equally to this work

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

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 implementation for the strongly typed functional language Koka [1]. The implementation supports typed algebraic effect handlers 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 readily 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 counting 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. Optimizing 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 *precise* and hold on to objects too long. This increases memory usage and prevents aggressive optimization of many reference 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, functional language using immutable data types together with a strong type and effect system. For concurrency, we precisely track when objects can become thread-shared (Section 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 specialization* (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 collector, our design has two important mitigations. First, *(co)inductive* 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 references 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 contribution 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 function can be modeled semantically as a mathematically *total* function, which always terminates without raising an exception (or having any other observable side effect). Effectful functions get more interesting effect types, like:

```
fun println( s : string ) : console ()
fun divide( 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:

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

Many compilers emit code similar to:

```
fun foo() {
  val xs = list(1,1000000)
  val ys = map(xs, inc)
  print(ys)
  drop(xs)
  drop(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⟨T⟩` (calling the destructor at the end of the scope), Rust's `Rc⟨T⟩` (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:

```
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 `map` 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:

```
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

$$\begin{tikzcd} (a) \arrow[d] \arrow[dd, bend right] & & \\ (b) \arrow[r] & (c) \arrow[r] & (d) \\ (e) \arrow[r] & (f) \arrow[r] & (g) \end{tikzcd}$$

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

(b) dup/drop insertion (2.2)

```
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 }
} }
```

(c) drop specialization (2.3)

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

(d) push down dup and fusion (2.3)

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

(f) drop-reuse specialization (2.4)

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

(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, lx, kx, vx, rx), ky, vy, ry)
      -> Node(Red, Node(Black, lx, kx, vx, rx), ky, vy,
                   Node(Black, ry, 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
```

```
void inorder( tree* root, void (*f)(tree* t) ) {
  tree* cursor = root;
  while (cursor != NULL /* Tip */) {
    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 = cursor;
        cursor = cursor->left;
      } else {
        // already set, restore
        f(cursor->value);
        pre->right = NULL;
        cursor = cursor->right;
} } } }
```

Fig. 2. Morris in-order tree traversal algorithm in C.

```
      -> if (k < kx) then Node(Red, ins(l, k, v), kx, vx, r)
         ...
    Node(Black, l, kx, vx, r)
      -> if (k < kx && is-red(l))
           then bal-left(ins(l,k,v), kx, vx, r)
         ...
}
```

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:

```
Node(Red, l, kx, vx, r) { // second branch
  val ru = if (is-unique(t)) then &t
           else { dup(l); dup(kx); dup(vx); dup(r); NULL }
  if (dup(k) < dup(kx)) {
    val y = ins(l,k,v)
    if (ru!=NULL) then { ru->left := y; ru } // fast path
                  else Node(Red, y, 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.

```
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(l,x,r) -> tmap(f,l,BinR(r,x,visit),Down) // A
      Tip        -> tmap(f,Tip,visit,Up)           // B
    }
    Up -> match(visit) {  // go up through the visitor
      Done       -> t                              // C
      BinR(r,x,v) -> tmap(f,r,BinL(t,f(x),v),Down) // D
      BinL(l,x,v) -> tmap(f,Bin(l,x,t),v,Up)       // E
} } }
```

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:

```
type tree {
  Tip
  Bin( left: tree, value : int, right: tree )
}
fun tmap( t : tree, f : int -> int ) : tree {
  match(t) {
    Bin(l,x,r) -> Bin( tmap(l,f), f(x), tmap(r,f) )
    Tip        -> Tip
} }
```

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* data 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 `tmap(f, 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 `D` (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 `tmap` 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 `dup` and `drop` operations. However, it is in tension with functions that have non-linear control flow, e.g. may throw an exception, use a `longjmp`, or create an asynchronous continuation that is never resumed. For example, if we look at the code for `map` before applying optimizations, we have:

```
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 `map`, then `xx` and `f` would leak and never be dropped. This is one reason why a C++ `shared_ptr` is tied to lexical scope; it integrates nicely with the stack unwinding mechanism for exceptions that guarantees each `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 *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 `Ok`, or `Error` if an exception is thrown. This is checked and propagated at every invocation[3].
For example, for `map` the compiled code (before optimization) becomes like:

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx); drop(xs); dup(f)
      match(f(x)) {
        Error(err) -> { drop(xx); drop(f); Error(err); }
        Ok(y) -> { match(map(xx, f)) {
          Error(err) -> drop(y); Error(err)
          Ok(ys)     -> 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 `error` values as these are implemented as *value* types which are not heap allocated.
This is similar to error handling in Swift [18] (although it requires the programmer to insert a `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 $\mu x.\ \mathsf{F}$ as a list of the derivative of that type, namely $\mathsf{list}\ (\frac{\partial}{\partial x}\ \mathsf{F}\ |_{x\ =\mu x.\mathsf{F}})$. In our case, calculating the derivative of the inductive `tree`, we get $\mu x.\ 1\ +\ (\mathit{tree} \times \mathit{int} \times x)\ +\ (\mathit{tree} \times \mathit{int} \times x)$, 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 `tshare : forall a. a -> io ()` 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 `drop` and `dup` operations can be implemented efficiently by avoiding atomic operations in the fast path by checking the thread-shared flag.
For example, `drop` may be implemented in C as:

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

However, this may still present quite some overhead as many `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 *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[4]:

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

The `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 *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 `ref`, dereferenced with `(!)` and updated using `(:=)`:

```
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 `st⟨h⟩` in some heap `h`. A reference cell of type `ref⟨h,a⟩` is a first-class *value* that contains a reference to a value of type `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 *race* with a read operation to update the reference counts. The pseudocode implementation of both operations is:

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

The read operation `(!)` first reads the current reference in `x`, and then increments its reference count. Suppose though that before the `dup`, the thread is suspended and another thread writes to the same reference: it will read the same object into `y`, update the reference, and then drop `y` – and if `y` has a reference count of 1 it will be freed! When the other thread resumes, it will now try to `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 *inductive* or *coinductive*. It can be shown that such data types are never cyclic (and functions that recurse over such data types always terminate).

[4] Since the thread-shared sign-bit is *stable*, we can do the test `b->header.cr <= 1` without needing expensive atomic operations and can use a `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, λ^1, which is closely based on linear logic. The operational semantics of λ^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 λ^1 and thus provide a theoretic foundation for Perceus.

3.1 Syntax

Figure 4 defines the syntax of our linear resource calculus λ^1. It is essentially an untyped lambda calculus extended with explicit binding as $\mathsf{val}\ x\ =\ e_1;\ e_2$, and pattern matching as match. 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 Δ, Γ are *multisets* containing variable names. We use the compact comma notation for summing (or splitting) multisets. For example, (Γ, x) adds x to Γ, and $(\Gamma_1,\ \Gamma_2)$ appends two multisets Γ_1 and Γ_2. The set of free variables of an expression e is denoted by $\mathsf{fv}(e)$, and the set of bound variables of a pattern p by $\mathsf{bv}(p)$.

3.2 The Linear Resource Calculus

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

Expressions

$$
\begin{array}{lcll}
e & ::= & v \mid e\ e & \text{(value, application)} \\
 & \mid & \mathsf{val}\ x\ =\ e;\ e & \text{(bind)} \\
 & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} & \text{(match)} \\
 & \mid & \mathsf{dup}\ x;\ e & \text{(duplicate)} \\
 & \mid & \mathsf{drop}\ x;\ e & \text{(drop)} \\
 & \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \} & \text{(match expr)} \\
v & ::= & x \mid \lambda x.\ e & \text{(variables, functions)} \\
 & \mid & C\ v_1\ \ldots\ v_n & \text{(constructor of arity } n\text{)} \\
p & ::= & C\ b_1 \ldots b_n & \text{(pattern)} \\
b & ::= & x \mid \_ & \text{(binder or wildcard)}
\end{array}
$$

Values

$$
\begin{array}{lcll}
v & ::= & x & \text{(variables, } f,\ y,\ z\text{)} \\
 & \mid & \lambda x.\ e & \text{(abstraction)} \\
 & \mid & C\ v_1\ \ldots\ v_n & \text{(constructor of arity } n\text{)}
\end{array}
$$

Patterns:

$$
\begin{array}{lcll}
p & ::= & C\ b_1 \ldots b_n & \text{(constructor of arity } n\text{)} \\
b & ::= & x \mid \_ & \text{(binder or wildcard)}
\end{array}
$$

Contexts $\quad \Delta,\ \Gamma\quad ::=\quad \varnothing\ \mid \Delta \cup x$

Syntactic shorthands

$$
\begin{array}{lcll}
e_1;\ e_2 & \triangleq & \mathsf{val}\ x\ =\ e_1;\ e_2 & \text{sequence, } x \notin \mathsf{fv}(e_2) \\
\lambda\_.\ e & \triangleq & \lambda x.\ e & x \notin \mathsf{fv}(e) \\
\lambda x.\ e & \triangleq & \lambda^{ys} x.\ e & ys\ =\ \mathsf{fv}(e)
\end{array}
$$

Fig. 4. Syntax of the linear resource calculus λ^1.

The key idea of λ^1 is that each resource (i.e., owned variable) is consumed *exactly* once. That is, a resource needs to be explicitly duplicated (in rule DUP) if it is needed more than once; or be explicitly dropped (in rule DROP) if it is not needed. The rules are closely related to linear typing.

Following the key idea, the variable rule 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 DROP to be able to discard y, which gives $\lambda x\ y.\ \mathsf{drop}\ y;\ x$.

The APP rule splits the owned environment Γ into two separate contexts Γ_1 and Γ_2 for expression e_1 and e_2 respectively. Each expression then consumes its corresponding owned environment. Since Γ_2 is consumed in the e_2 derivation, we know that resources in Γ_2 are surely alive when deriving e_1, and thus we can *borrow* Γ_2 in the e_1 derivation. The rule is quite similar to the [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

$$\boxed{\begin{array}{cccccc} \Delta \mid & \Gamma & \vdash & e & \rightsquigarrow & e' \\ \uparrow & \uparrow & & \uparrow & & \downarrow \end{array}}$$

$$\frac{}{\Delta \mid x \vdash x \rightsquigarrow x}\;[\textsc{var}]$$

$$\frac{\Delta \mid \Gamma, x \vdash e \rightsquigarrow e' \quad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash e \rightsquigarrow \mathsf{dup}\; x;\; e'}\;[\textsc{dup}]$$

$$\frac{\Delta \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow \mathsf{drop}\; x;\; e'}\;[\textsc{drop}]$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \quad \Delta \mid \Gamma_2 \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \vdash e_1\; e_2 \rightsquigarrow e_1'\; e_2'}\;[\textsc{app}]$$

$$\frac{\varnothing \mid ys, x \vdash e \rightsquigarrow e' \quad ys = \mathsf{fv}(\lambda x.\, e)}{\Delta \mid ys \vdash \lambda x.\, e \rightsquigarrow \lambda^{ys}\, x.\, e'}\;[\textsc{lam}]$$

$$\frac{\begin{array}{l} x \notin \Delta, \Gamma_1, \Gamma_2 \\ \Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \quad \Delta \mid \Gamma_2, x \vdash e_2 \rightsquigarrow e_2' \end{array}}{\Delta \mid \Gamma_1, \Gamma_2 \vdash \mathsf{val}\; x = e_1;\; e_2 \rightsquigarrow \mathsf{val}\; x = e_1';\; e_2'}\;[\textsc{bind}]$$

$$\frac{\Delta \mid \Gamma, \mathsf{bv}(p_i) \vdash e_i \rightsquigarrow e_i'}{\Delta \mid \Gamma, x \vdash \mathsf{match}\; x \,\{\, \overline{p_i \mapsto e_i} \,\} \rightsquigarrow \mathsf{match}\; x \,\{\, \overline{p_i \mapsto e_i'} \,\}}\;[\textsc{match}]$$

$$\frac{\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \vdash v_i \rightsquigarrow v_i' \quad 1 \leqslant i \leqslant n}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash C\; v_1 \ldots v_n \rightsquigarrow C\; v_1' \ldots v_n'}\;[\textsc{con}]$$

Fig. 5. Declarative linear resource rules of λ^1.

$$\begin{array}{lll} \mathsf{E} & ::= & \square \mid \mathsf{E}\; e \mid v\; \mathsf{E} \\ & \mid & \mathsf{val}\; x = \mathsf{E};\; e \end{array} \qquad \frac{e \longrightarrow e'}{\mathsf{E}[e] \longmapsto \mathsf{E}[e']}\;[\textsc{eval}]$$

$$\begin{array}{llll} (app) & (\lambda x.\, e)\; v & \longrightarrow & e[x{:=}v] \\ (bind) & \mathsf{val}\; x = v\,;\; e & \longrightarrow & e[x{:=}v] \\ (match) & \mathsf{match}\; (C\; v_1 \ldots v_n)\; \{\overline{p_i \to e_i}\} & & \\ & & \longrightarrow & e_i[x_1{:=}v_1, \ldots, x_n{:=}v_n] \\ & & & \text{with } p_i = C\; x_1 \ldots x_n \end{array}$$

Fig. 6. Standard strict semantics for λ^1.

in $\lambda f\; g\; x.\; \mathsf{dup}\; x;\; (f\; x)\; (g\; x)$. With the borrowing environment it is now possible to derive a translation with the dup right before passing x to f: $\lambda f\; g\; x.\; (f\; (\mathsf{dup}\; x;\; x))\; (g\; x))$. Notice rule DUP allows dup from the borrowing environment, where DROP only applies to the owned environment.

The LAM rule is interesting as it essentially derives the body of the lambda independently. The premise $ys = \mathsf{fv}(\lambda x.e)$ requires that exactly the free variables in the lambda are owned – this corresponds to the notion that a lambda is allocated as a closure at runtime that holds all free variables of the lambda (and thus the lambda expression *consumes* the free variables). The body of a lambda is evaluated only when 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 ys, as $\lambda^{ys} x.\; e$, 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 BIND rule is similar to application and borrows Γ_2 in the derivation for the bound expression. This is the main reason to not consider $\mathsf{val}\; x = e_1;\; e_2$ as syntactic sugar for $(\lambda x.\; e_2)\; e_1$. The MATCH rule consumes the scrutinee and owns the bound variables in each pattern for each branch. For constructors (rule 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 $\lceil e \rceil$ to erase all drop and dup in the expression e. We can now state that derivations leave expressions unchanged except for inserting dup/drop operations: if $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$ then $e = \lceil e' \rceil$.

Lemma 1. (*Translation only inserts dup/drop*)
If $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$ then $e = \lceil e' \rceil$.

3.3 Semantics

Figure 6 defines standard semantics for λ^1 using strict evaluation contexts [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 $(match)$ replies on the internal form of expression $\mathsf{match}\; e\; \{\; \overline{p_i \to e_i}\; \}$: after substitution (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 $x \mapsto^n v$ points to a value v with a reference count of n (with $n \geqslant 1$). In these semantics, values other than variables are allocated in the heap with rule (lam_r) and rule (con_r). The evaluation rules discard entries from the heap when the reference count drops to zero. Any allocated lambda is annotated as $\lambda^{ys} x.\; e$ to clarify that these are essentially *closures* holding an environment ys and a code pointer $\lambda x.\; e$. Note that it is important that the environment ys is a multi-set. After the initial translation, ys will be equivalent to the free variables in the body (see rule 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 (app_r) needs to satisfy the assumptions made when deriving the abstraction in rule LAM. First, the (app_r) rule inserts dup to duplicate variables ys, as these are owned in rule LAM. It then drops the reference to the closure itself. Rule $(match_r)$ is similar to

$$\begin{array}{lll} \mathsf{H} & : & x \rightarrow (\mathbb{N}^{+}, v) \\ \mathsf{E} & ::= & \square \mid \mathsf{E}\ e \mid x\ \mathsf{E} \mid \mathsf{val}\ x = \mathsf{E};\ e \\ & \mid & C\ x_1 \ldots x_i\ \mathsf{E}\ v_j \ldots v_n \end{array} \qquad \frac{\mathsf{H} \mid e \longrightarrow_r \mid \mathsf{H}'\ e'}{\mathsf{H} \mid \mathsf{E}[e] \longmapsto_r \mathsf{H}' \mid \mathsf{E}[e']}\ [\textsc{eval}]$$

$$\begin{array}{llllll}
(lam_r) & \mathsf{H} \mid (\lambda^{ys}\ x.\ e) & \longrightarrow_r & \mathsf{H}, f \mapsto^1 \lambda^{ys} x.\ e & \mid f & \text{fresh } f \\
(con_r) & \mathsf{H} \mid C\ x_1 \ldots x_n & \longrightarrow_r & \mathsf{H}, z \mapsto^1 C\ x_1 \ldots x_n & \mid z & \text{fresh } z
\end{array}$$

$$\begin{array}{lllll}
(app_r) & \mathsf{H} \mid f\ z & \longrightarrow_r & \mathsf{H} \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ f;\ e[x{:=}z] & (f \mapsto^n \lambda^{ys} x.\ e) \in \mathsf{H} \\
(match_r) & \mathsf{H} \mid \mathsf{match}\ x\ \{\overline{p_i \rightarrow e_i}\} & \longrightarrow_r & \mathsf{H} \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ x;\ e_i[xs{:=}ys] & \text{with } p_i = C\ xs \text{ and } (x \mapsto^n C\ ys) \in \mathsf{H} \\
(bind_r) & \mathsf{H} \mid \mathsf{val}\ x = y\ ;\ e & \longrightarrow_r & \mathsf{H} \mid e[x{:=}y] &
\end{array}$$

$$\begin{array}{llllll}
(dup_r) & \mathsf{H}, x \mapsto^n v & \mid \mathsf{dup}\ x;\ e & \longrightarrow_r & \mathsf{H}, x \mapsto^{n+1} v \mid e & \\
(drop_r) & \mathsf{H}, x \mapsto^{n+1} v & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H}, x \mapsto^n v \mid e & \text{if } n \geqslant 1 \\
(dlam_r) & \mathsf{H}, x \mapsto^1 \lambda^{ys} z.e & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H} \mid \mathsf{drop}\ ys;\ e & \\
(dcon_r) & \mathsf{H}, x \mapsto^1 C\ ys & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H} \mid \mathsf{drop}\ ys;\ e &
\end{array}$$

Fig. 7. Reference-counted heap semantics for λ^1.

rule (app_r), which duplicates the newly bound pattern bindings and drops the scrutinee. Rule $(bind_r)$ simply substitutes the bound variable x with the resource y.

Duping a resource is straightforward as rule (dup_r) merely increments the reference count of the resource. Dropping is more involved. Rule $(drop_r)$ just decrements the reference count when there are still multiple copies of it. But when the reference count would drop to zero, rule $(dlam_r)$ and rule $(dcon_r)$ actually *free* a heap entry and then dynamically insert drop operations to drop their fields recursively.

The tricky part of the reference counting semantics is showing *correctness*. We prove this in two parts. First, we prove that the reference counting semantics is *sound* and corresponds to the standard semantics. Below we use heaps as substitutions on expressions. We write $[\mathsf{H}]e$ to mean H applied as a substitution to expression e.

Theorem 1. (*Reference-counted heap semantics is sound*)
If $\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$ and $e \longmapsto^* v$, then also $\varnothing \mid e' \longmapsto^*{}_r\ \mathsf{H} \mid x$ with $[\mathsf{H}]x = v$.

To prove this theorem we need to maintain strong invariants at each evaluation step to ensure a variable is still alive if it is going to be referred later. The proof can be found in Appendix D.2. Second, we prove that the reference counting semantics never *hold on* to unused variables. We first define the notion of *reachability*.

Definition 1. (*Reachability*)
We say a variable x is reachable in terms of a heap H and an expression e, denoted as $\mathsf{reach}(x,\ \mathsf{H} \mid e)$, if (1) $x \in \mathsf{fv}(e)$; or (2) for some y, we have $\mathsf{reach}(y,\ \mathsf{H} \mid e) \wedge y \mapsto^n v \in \mathsf{H} \wedge \mathsf{reach}(x,\ \mathsf{H} \mid v)$.

With reachability, we can formally show:

Theorem 2. (*Reference counting leaves no garbage*)
Given $\varnothing; \varnothing \vdash e \rightsquigarrow e'$, and $\varnothing \mid e' \longmapsto^*{}_r\ \mathsf{H} \mid x$, then for every intermediate state $\mathsf{H}_i \mid e_i$, we have that for all $y \in \mathsf{dom}(\mathsf{H}_i)$, $\mathsf{reach}(y,\ \mathsf{H}_i \mid e_i)$.

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, λ^1 does not model *mutable references* (Section 2.7.3). From Theorem 2 we see that mutable references are indeed the only source of cycles. A natural extension of the system is to include mutable references and thus cycles. In that case, we could generalize Theorem 2, where the conclusion would be that for all resource in the heap, it is either reachable from the expression, or it is part of a cycle.

The above theorems establish the correctness of the reference-counted heap semantics. However, correctness does not imply *precision*, ie. that the heap is *garbage free*. Eventually all live data is discarded but it may well hold on to live data too long by delaying drop operations. As an example, consider $y \mapsto^1 () \mid (\lambda x.\ x)\ (\mathsf{drop}\ y;\ ())$, where y is reachable but dropped too late: it is only dropped after the lambda gets allocated. In contrast, a *garbage free* algorithm would produce $y \mapsto^1 () \mid \mathsf{drop}\ y;\ (\lambda x.\ x)\ ()$. In the next section we present Perceus as a syntax directed algorithm of the linear resource calculus and show that it is *garbage free*.

3.4 Perceus

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

$$\boxed{\underset{\uparrow}{\Delta} \mid \underset{\uparrow}{\Gamma} \vdash_{\mathsf{s}} \underset{\uparrow}{e} \rightsquigarrow \underset{\downarrow}{e'} \qquad \Delta \cap \Gamma = \varnothing \quad \Gamma \subseteq \mathsf{fv}(e) \quad \mathsf{fv}(e) \subseteq \Delta, \Gamma \quad \text{multiplicity of each member in } \Delta, \Gamma \text{ is } 1}$$

$$\frac{}{\Delta \mid x \vdash_{\mathsf{s}} x \rightsquigarrow x}\ [\textsc{svar}] \qquad \frac{}{\Delta, x \mid \varnothing \vdash_{\mathsf{s}} x \rightsquigarrow \mathsf{dup}\ x;\ x}\ [\textsc{svar-dup}]$$

$$\frac{\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \rightsquigarrow e_1' \quad \Delta \mid \Gamma_2 \vdash_{\mathsf{s}} e_2 \rightsquigarrow e_2' \quad \Gamma_2 = \Gamma \cap \mathsf{fv}(e_2)}{\Delta \mid \Gamma \vdash_{\mathsf{s}} e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\ [\textsc{sapp}]$$

$$\frac{\begin{array}{l} x \in \mathsf{fv}(e) \quad \varnothing \mid ys, x \vdash_{\mathsf{s}} e \rightsquigarrow e' \\ ys = \mathsf{fv}(\lambda x.\ e) \quad \Delta_1 = ys - \Gamma \end{array}}{\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \lambda x.\ e \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ e'}\ [\textsc{slam}] \qquad \frac{\begin{array}{l} x \notin \mathsf{fv}(e) \quad \varnothing \mid ys \vdash_{\mathsf{s}} e \rightsquigarrow e' \\ ys = \mathsf{fv}(\lambda x.\ e) \quad \Delta_1 = ys - \Gamma \end{array}}{\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \lambda x.\ e \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ (\mathsf{drop}\ x;\ e')}\ [\textsc{slam-drop}]$$

$$\frac{\begin{array}{l} x \in \mathsf{fv}(e_2) \quad x \notin \Delta, \Gamma \\ \Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \rightsquigarrow e_1' \\ \Delta \mid \Gamma_2, x \vdash_{\mathsf{s}} e_2 \rightsquigarrow e_2' \quad \Gamma_2 = \Gamma \cap (\mathsf{fv}(e_2) - x) \end{array}}{\Delta \mid \Gamma \vdash_{\mathsf{s}} \mathsf{val}\ x = e_1;\ e_2 \rightsquigarrow \mathsf{val}\ x = e_1';\ e_2'}\ [\textsc{sbind}] \qquad \frac{\begin{array}{l} x \notin \mathsf{fv}(e_2),\ \Delta,\ \Gamma \\ \Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \rightsquigarrow e_1' \\ \Delta \mid \Gamma_2 \vdash_{\mathsf{s}} e_2 \rightsquigarrow e_2' \quad \Gamma_2 = \Gamma \cap \mathsf{fv}(e_2) \end{array}}{\Delta \mid \Gamma \vdash_{\mathsf{s}} \mathsf{val}\ x = e_1;\ e_2 \rightsquigarrow \mathsf{val}\ x = e_1';\ \mathsf{drop}\ x;\ e_2'}\ [\textsc{sbind-drop}]$$

$$\frac{\Delta \mid \Gamma_i \vdash_{\mathsf{s}} e_i \rightsquigarrow e_i' \quad \Gamma_i = (\Gamma, \mathsf{bv}(p_i)) \cap \mathsf{fv}(e_i) \quad \Gamma_i' = (\Gamma, \mathsf{bv}(p_i)) - \Gamma_i}{\Delta \mid \Gamma, x \vdash_{\mathsf{s}} \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \rightsquigarrow \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{drop}\ \Gamma_i';\ e_i'}\ \}}\ [\textsc{smatch}]$$

$$\frac{\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \vdash_s v_i \rightsquigarrow v_i' \quad 1 \leqslant i \leqslant n \quad \Gamma_i = (\Gamma - \Gamma_{i+1} - \ldots - \Gamma_n) \cap \mathsf{fv}(v_i)}{\Delta \mid \Gamma \vdash_s C\ v_1 \ldots v_n \rightsquigarrow C\ v_1' \ldots v_n'}\ [\textsc{scon}]$$

Fig. 8. Syntax-directed linear resource rules of λ^1.

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 Γ. We pass the intersection of Γ 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 $\Gamma \subseteq \mathsf{fv}(\lambda x.e)$ and thus $\Gamma \subseteq ys$. If there are any free variables not in Γ, they must be part of the borrowed environment (as Δ_1) 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 to be dropped as they no longer occur as free variables in that branch. The owned environment Γ_i in the ith branch is the intersection of $(\Gamma, \mathsf{bv}(p_i))$ and the free variables in that branch; any other owned variables (as Γ_i') are dropped at the start of the branch. Rule scon deterministically splits the environment Γ 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 $\Delta \mid \Gamma \vdash_{\mathsf{s}} e \rightsquigarrow e'$ then also $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$.

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 $\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$ and $\varnothing \mid e' \longmapsto^*_r \mathsf{H} \mid x$, then for every intermediate state $\mathsf{H}_i \mid e_i$ that is not at a dup/drop operation ($e_i \neq \mathsf{E}[\mathsf{drop}\ x;\ e_i']$ and $e_i \neq \mathsf{E}[\mathsf{dup}\ x;\ e_i']$), we have that for all $y \in \mathsf{dom}(\mathsf{H}_i)$, $\mathsf{reach}(y,\ \mathsf{H}_i \mid \lceil e_i \rceil)$.

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, $y \mapsto^1 () \mid (\lambda x.\ x)\ (\mathsf{drop}\ y;\ ())$ 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.

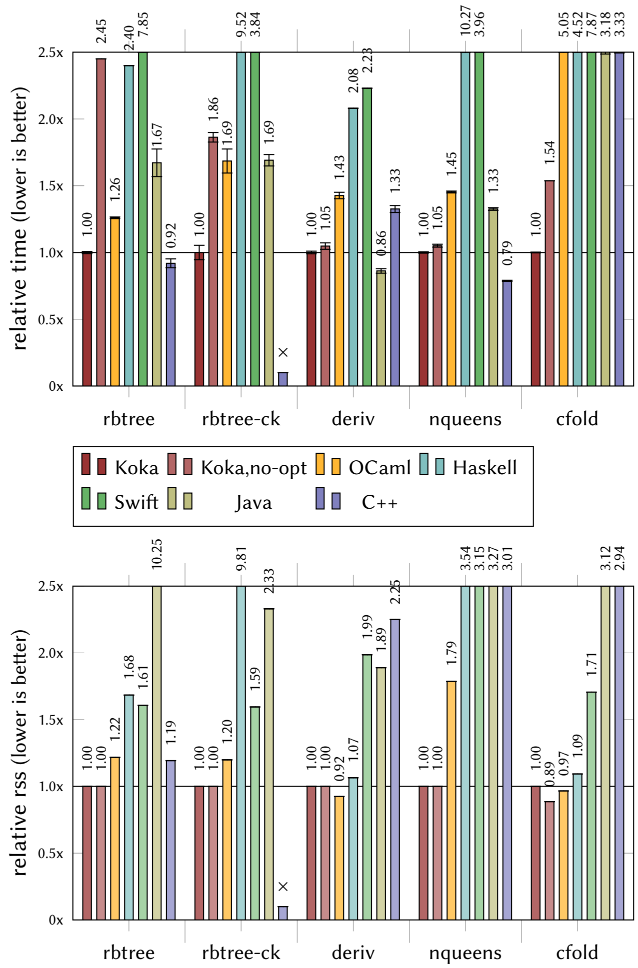

Fig. 9. Relative execution time and peak working set with respect to Koka. Using a 6-core 64-bit AMD 3600XT 3.8Ghz with 64GiB 3600Mhz memory, Ubuntu 20.04.

4 Benchmarks

In this section we discuss initial benchmarks of Perceus as implemented in Koka, versus state-of-the-art memory reclamation implementations in various other languages. Since we compare across languages we need to interpret the results with care – the results depend not only on memory reclamation but also on the different optimizations performed by each compiler and how well we can translate each benchmark to that particular language. We view these results therefore mostly as *evidence that the Perceus reference counting technique is viable and can be competitive* and *not* as a direct comparison of absolute performance between systems.

As such, we selected only benchmarks that stress memory allocation, and we tried to select mature comparison systems that use a range of memory reclamation techniques and are considered best-in-class. The systems we compare are:

- Koka 2.0.3, compiling the generated C code with gcc 9.3.0 using a customized version of the mimalloc allocator [24]. 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 partially 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 `nqueens`, 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 λ^1 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 Petrank [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 λ^1, 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

[1] Koka repository. 2019. URL https://github.com/koka-lang/koka.

[2] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. Technical Report UCB/ERL M524, EECS Department, University of California, Berkeley, Jun 1975. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/1975/29109.html.

[3] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6: 6–2, 1996.

[4] Jiho Choi, Thomas Shull, and Josep Torrellas. Biased reference counting: Minimizing atomic operations in garbage collection. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, 2018. doi:10.1145/3243176.3243195.

[5] George E Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3 (12): 655–657, 1960.

[6] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. In *in Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.

[7] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19 (9): 522–526, September 1976. ISSN 0001-0782. doi:10.1145/360336.360345.

[8] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL)*, pages 113–123. ACM press, January 1993.

[9] Gaspard Férey and Natarajan Shankar. Code generation using a formal model of reference counting. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods*, pages 150–165. Springer International Publishing, 2016. ISBN 978-3-319-40648-0.

[10] Free Software Foundation, Silicon Graphics, and Hewlett–Packard Company. Internal red-black tree implemention for "stl::map". URL https://code.woboq.org/gcc/libstdc++-v3/src/c++98/tree.cc.html.

[11] Matt Gallagher. Reference counted releases in Swift. Blog post, December 2016. URL https://www.cocoawithlove.com/blog/resources-releases-reentrancy.html.

[12] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. In *8th International Symposium on Parallel Architectures,Algorithms and Networks (ISPAN'05)*, 2005.

[13] Clemens Grelck and Kai Trojahner. Implicit memory management for SAC. In *6th International Workshop on Implementation and Application of Functional Languages (IFL'04)*, September 2004.

[14] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21. IEEE, 1978.

[15] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, page 12–23. ACM, 1995. ISBN 0897917197. doi:10.1145/224164.224173.

[16] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, page 300–314. ACM, 1985. ISBN 0897911474. doi:10.1145/318593.318660.

[17] Gérard P. Huet. The zipper. *Journal of Functional Programming*, 7 (5): 549–554, 1997.

[18] Apple Inc. The Swift guide: Error handling. 2017. URL https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html.

[19] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0201896834.

[20] Daan Leijen. Koka: Programming with row polymorphic effect types. In *MSFP'14, 5th workshop on Mathematically Structured Functional Programming*, 2014. doi:10.4204/EPTCS.153.8.

[21] Daan Leijen. Algebraic effects for functional programming. Technical Report MSR-TR-2016-29, Microsoft Research technical report, August 2016. Extended version of [22].

[22] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, pages 486–499, January 2017a. ISBN 978-1-4503-4660-3. doi:10.1145/3009837.3009872.

[23] Daan Leijen. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 16–29, 2017b. ISBN 978-1-4503-5183-6. doi:10.1145/3122975.3122977.

[24] Daan Leijen, Zorn Ben, and Leo de Moura. Mimalloc: Free list sharding in action. *Programming Languages and Systems*, 11893, 2019. doi:10.1007/978-3-030-34175-6_13. APLAS'19.

[25] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28 (1): 1–69, January 2006. ISSN 0164-0925. doi:10.1145/1111596.1111597.

[26] Prabhaker Mateti and Ravi Manghirmalani. Morris' tree traversal algorithm reconsidered. *Science of Computer Programming*, 11 (1): 29–43, 1988. ISSN 0167-6423. doi:10.1016/0167-6423(88)90063-9.

[27] Conor McBride. The derivative of a regular type is its type of one-hole contexts, 2001. URL http://strictlypositive.org/diff.pdf. (Extended Abstract).

[28] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3 (4): 184–195, 1960.

[29] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. SISAL: streams and iteration in a single-assignment language. language reference manual, version 1. 1. Technical Report LLL/M-146, ON: DE83016576, Lawrence Livermore National Lab., CA, USA, 7 1983.

[30] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15 (6): 491–504, June 2004. doi:10.1109/TPDS.2004.8.

[31] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. 2012. ISBN 978-1449323912. URL https://dev.realworldocaml.org.

[32] Joseph M. Morris. Traversing binary trees simply and cheaply. *Information Processing Letters*, 9 (5): 197 – 200, 1979. doi:10.1016/0020-0190(79)90068-1.

[33] Chris Okasaki. *Purely Functional Data Structures*. Colombia University, June 1999a. ISBN 9780521663502.

[34] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9 (4): 471–477, 1999b. doi:10.1017/S0956796899003494.

[35] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32 (1): 3 – 47, 1998. doi:10.1016/S0167-6423(97)00029-4.

[36] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11 (1): 69–94, 2003. doi:10.1023/A:1023064908962.

[37] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. volume 9, 2013. doi:10.2168/LMCS-9(4:23)2013.

[38] Sven-Bodo Scholz. Single Assignment C: Efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13 (6): 1005–1059, November 2003. doi:10.1017/S0956796802004458.

[39] Wolfram Schulte. Deriving residual reference count garbage collectors. In Manuel Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP)*, pages 102–116, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48695-4.

[40] Herb S. Sutter. Zero-overhead deterministic exceptions: Throwing values. C++ open-std proposal P0709 R2, 10 2018.

[41] David N. Turner and Phillip Wadler. Operational interpretations of linear logic. (227): 231–248, 1999.

[42] Sebastian Ullrich and Leonardo de Moura. Counting immutable beans – reference counting optimized for purely functional programming. In *Proceedings of the 31st symposium on Implementation and Application of Functional Languages (IFL'19)*, September 2019.

[43] David Ungar, David Grove, and Hubertus Franke. Dynamic atomicity: Optimizing swift memory management. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, DLS 2017, page 15–26, 2017. doi:10.1145/3133841.3133843.

[44] Phillip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

[45] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115 (1): 38–94, November 1994. doi:10.1006/inco.1994.1093.

[46] Ningning Xie and Daan Leijen. Effect handlers in Haskell, evidently. In *Proceedings of the 2020 ACM SIGPLAN Symposium on Haskell*, Haskell'20, August 2020.

[47] Ningning Xie, Jonathan Brachthäuser, Phillip Schuster, Daniel Hillerström, and Daan Leijen. Effect handlers, evidently. In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'2020)*, ICFP '20, August 2020.

[48] Danil Yarantsev. Orc - nim's cycle collector. October 2020. URL `https://nim-lang.org/blog/2020/10/15/introduction-to-arc-orc-in-nim.html`.

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 naive Koka compiler is not (yet) doing. It is also interesting to see that the "no-opt" Koka is only a bit slower than

```
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(_, ly, ky, vy, Node(Red, lx, kx, vx, rx))
      -> Node(Red, Node(Black, ly, ky, vy, lx), kx, vx,
                   Node(Black, rx, k, v, r))
    Node(_, lx, kx, vx, rx)
      -> Node(Black, Node(Red, lx, kx, vx, rx), k, v, r)
} }

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

fun ins(t: tree, k: int, v: bool): tree {
  match(t) {
    Leaf -> Node(Red, Leaf, k, v, Leaf)
    Node(Red, l, kx, vx, r)
      -> if (k < kx) then Node(Red, ins(l, k, v), kx, vx, r)
         elif (k == kx) then Node(Red, l, k, v, r)
         else Node(Red, l, kx, vx, ins(r, k, v))
    Node(Black, l, kx, vx, r)
      -> if (k < kx) then
           (if (is-red(l))
              then bal-left(ins(l,k,v), kx, vx, r)
              else Node(Black, ins(l, k, v), kx, vx, r))
         elif (k == kx) then Node(Black, l, k, v, r)
         elif (is-red(r)) then bal-right(l, kx, vx, ins(r,k,v))
         else Node(Black, l, kx, vx, ins(r, k, v))
} }

fun set-black(t: tree) : tree {
  match(t) {
    Node(_, l, k, v, r) -> Node(Black, l, k, v, r)
    _ -> 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

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.

- `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 `deriv`, for the C++ version we do *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.
- `cfold`: performs constant-folding over a large symbolic expression. This benchmark is similar to the `deriv` benchmark and manipulates a complex expression graph. Koka does significantly better than other systems. Just as in `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. `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 4.30GHz, Ubuntu 20.04.

Figure 12 show execution time and peak working sets on a 16-core AMD 5950X at 3.4Ghz.

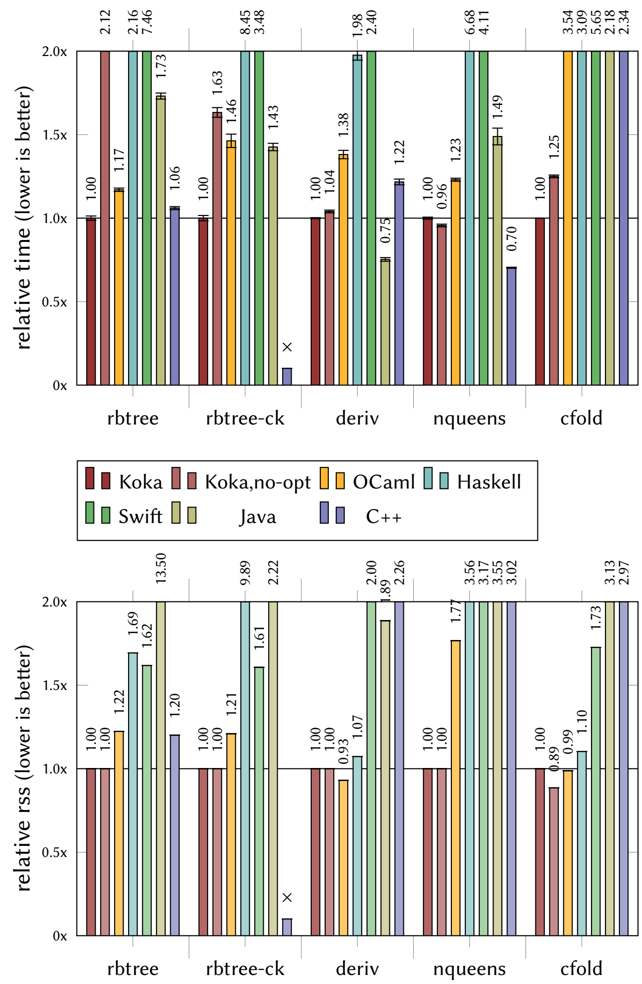

Fig. 11. Relative execution time and peak working set with respect to Koka (lower is better). On a 10-core Intel Core i9-7900X at 4.30GHz, Ubuntu 20.04.

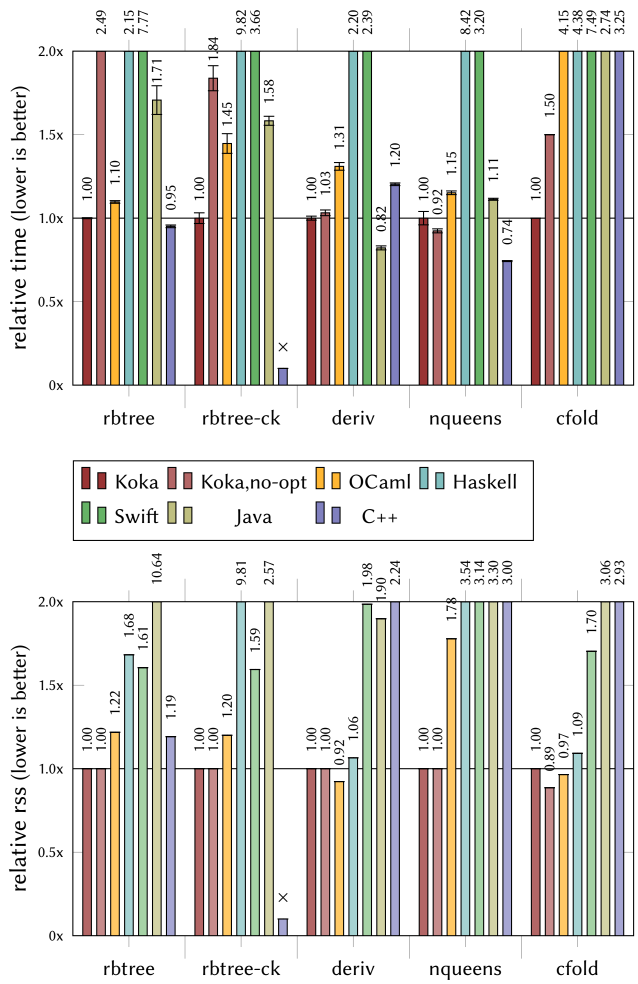

Fig. 12. Relative execution time and peak working set with respect to Koka (lower is better). On a 16-core AMD 5950X at 3.4Ghz, Ubuntu 20.04.

D Proofs

D.1 A Heap Reference Counting Calculus

$$\frac{\mathsf{H} \vdash v_1 \dashv \mathsf{H}_1 \ldots \mathsf{H}_{n-1} \vdash v_n \dashv \mathsf{H}_n}{\mathsf{H} \vdash C\ v_1 \ldots v_n \dashv \mathsf{H}_n}\ [\textsc{drcon}]$$

$$\frac{}{\mathsf{H},\ x \mapsto^{n+1} v \vdash x \dashv \mathsf{H},\ x \mapsto^{n} v}\ [\textsc{drvar}]$$

$$\frac{\mathsf{H} \vdash ys \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e \vdash x \dashv \mathsf{H}_1}\ [\textsc{drvarlam}]$$

$$\frac{\mathsf{H} \vdash ys \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} C\ ys \vdash x \dashv \mathsf{H}_1}\ [\textsc{drvarcon}]$$

$$\frac{\mathsf{H},\ x \mapsto^{n+1} v \vdash e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{n} v \vdash \mathsf{dup}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{drdup}]$$

$$\frac{\mathsf{H},\ x \mapsto^{n} v \vdash e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{n+1} v \vdash \mathsf{drop}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{drdrop}]$$

$$\frac{\mathsf{H} \vdash \mathsf{drop}\ ys;\ e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} C\ ys \vdash \mathsf{drop}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{drdropcon}]$$

$$\frac{\mathsf{H} \vdash \mathsf{drop}\ ys;\ e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e \vdash \mathsf{drop}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{drdroplam}]$$

$$\frac{\mathsf{H} \vdash ys \dashv \mathsf{H}_1 \quad ys \mapsto^{1} (),\ x \mapsto^{1} () \vdash e \dashv \varnothing}{\mathsf{H} \vdash \lambda^{ys}\ x.\ e \dashv \mathsf{H}_1}\ [\textsc{drlam}]$$

$$\frac{\mathsf{H} \vdash e_1 \dashv \mathsf{H}_1 \quad \mathsf{H}_1 \vdash e_2 \dashv \mathsf{H}_2}{\mathsf{H} \vdash e_1\ e_2 \dashv \mathsf{H}_2}\ [\textsc{drapp}]$$

$$\frac{\mathsf{H} \vdash e_1 \dashv \mathsf{H}_1 \quad \mathsf{H}_1, x \mapsto^{1} () \vdash e_2 \dashv \mathsf{H}_2 \quad x \notin \mathsf{H},\ \mathsf{H}_2}{\mathsf{H} \vdash \mathsf{val}\ x = e_1\ ;\ e_2 \dashv \mathsf{H}_2}\ [\textsc{drbind}]$$

$$\frac{\mathsf{H} \vdash x \dashv \mathsf{H}_1 \quad \mathsf{H}_1,\ [\![\mathsf{bv}(p_i)]\!] \vdash e_i \dashv \mathsf{H}' \quad \mathsf{bv}(p_i) \notin \mathsf{H}, \mathsf{H}'}{\mathsf{H} \vdash \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \dashv \mathsf{H}'}\ [\textsc{drmatch}]$$

Lemma 2. (*Heap Reference Counting Free variables*)
If $\mathsf{H}_1 \vdash e \dashv \mathsf{H}_2$, then $\mathsf{fv}(e) \in \mathsf{H}_1$, and $\mathsf{fv}(\mathsf{H}_2) \in \mathsf{H}_1$ with same domains.

Proof. (*Of Lemma 2*) By a straightforward induction on the rules. □

Definition 2. (*Extension*)
H is extended with x, denoted as $\mathsf{H} \mathbin{+\!\!+} x$, where $\mathbin{+\!\!+}$ works as follows:
(1) if $\mathsf{H} = \mathsf{H}',\ x \mapsto^{n} v$, then $\mathsf{H} \mathbin{+\!\!+} x = \mathsf{H}',\ x \mapsto^{n+1} ys$;
(2) if $x \notin \mathsf{H}$, then $\mathsf{H} \mathbin{+\!\!+} x = (\mathsf{H},\ x \mapsto^{1} v) \mathbin{+\!\!+} \mathsf{fv}(v)$.
We omit the domain of x in $\mathsf{H} \mathbin{+\!\!+} x$ 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 $(\mathsf{H},\ x \mapsto^1 v) +\!\!+\ \mathsf{fv}(v) = \mathsf{H} +\!\!+\ \mathsf{fv}(v),\ x \mapsto^1 v$ in (2).

Lemma 3. (*Drop is dual to extension*)
If $\mathsf{H}_1 \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}_2$, then $\mathsf{H}_1 = \mathsf{H}_2 +\!\!+\ x$. Similarly, if $\mathsf{H}_1 \vdash x \dashv \mathsf{H}_2$, then $\mathsf{H}_1 = \mathsf{H}_2 +\!\!+\ x$.

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

$\mathsf{H},\ x \mapsto^{n+1} ys \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H},\ x \mapsto^{n} ys$ DRDROP, DRCON

case

$\mathsf{H},\ x \mapsto^1 \lambda^{ys} z.\ e \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash \mathsf{drop}\ ys;\ () \dashv \mathsf{H}_1$	DRDROPLAM
$\mathsf{H} = \mathsf{H}_1 +\!\!+\ ys$	I.H.
$\mathsf{H}, x \mapsto^1 \lambda^{ys} z.e = \mathsf{H}_1 +\!\!+\ x$	by definition

case

$\mathsf{H},\ x \mapsto^1 C\ ys \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash \mathsf{drop}\ ys;\ () \dashv \mathsf{H}_1$	DRDROPCON
$\mathsf{H} = \mathsf{H}_1 +\!\!+\ ys$	I.H.
$\mathsf{H}, x \mapsto^1 C\ ys = \mathsf{H}_1 +\!\!+\ x$	by definition

□

Lemma 4. (*Extension is dual to drop*)
$\mathsf{H} +\!\!+\ x \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}$. Similarly, $\mathsf{H} +\!\!+\ x \vdash x \dashv \mathsf{H}$.

Proof. (*Of Lemma 4*) By induction on $+\!\!+x$.
case

$\mathsf{H} = \mathsf{H}',\ x \mapsto^n ys$	if
$\mathsf{H}+\!\!+x = \mathsf{H}',\ x \mapsto^{n+1} ys$	by definition
$\mathsf{H}',\ x \mapsto^{n+1} ys \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}$	DRDROP

case

$x \notin \mathsf{H}$	if
$ys = \mathsf{fv}(v)$	let
$\mathsf{H} +\!\!+\ x = \mathsf{H} +\!\!+\ ys,\ x \mapsto^1 v$	by definition
$\mathsf{H} +\!\!+\ ys \vdash ys \dashv \mathsf{H}$	I.H.
$\mathsf{H} +\!\!+\ ys,\ x \mapsto^1 v \vdash x \dashv \mathsf{H}$	DRVARLAM or DRVARCON

□

Lemma 5. (*Extension Commutativity*)
$\mathsf{H} +\!\!+\ x +\!\!+\ y = \mathsf{H} +\!\!+\ y +\!\!+\ x$.

Proof. (*Of Lemma 5*) By induction on $+\!\!+x$ and $+\!\!+y$, then we do case analysis. **case** $x \in \mathsf{H}$. Then $\mathsf{H} = \mathsf{H}',\ x \mapsto^n v$.
By definition, $\mathsf{H} +\!\!+\ x +\!\!+\ y = (\mathsf{H}', x \mapsto^{n+1} v) +\!\!+\ y$. Since the way $+\!\!+$ works only depends on whether x exists but not the exact number of its occurrence, we can decrease the number of x, do $+\!\!+y$ and then add x back. That is, $(\mathsf{H}', x \mapsto^{n+1} v) +\!\!+\ y = (\mathsf{H}',\ x \mapsto^n v) +\!\!+\ y +\!\!+\ x = \mathsf{H} +\!\!+\ y +\!\!+\ x$.
case $y \in \mathsf{H}$ is similar as the previous case.
case $x,\ y \notin \mathsf{H}$. Then $\mathsf{H} +\!\!+\ x = \mathsf{H} +\!\!+\ xs,\ x \mapsto^1 v$ where $xs = \mathsf{fv}(v)$.
subcase Assume $+\!\!+y$ won't cause $+\!\!+x$, then $+\!\!+y$ doesn't care about the existance of x.
So $(\mathsf{H} +\!\!+\ xs,\ x \mapsto^1 v) +\!\!+\ y = \mathsf{H} +\!\!+\ xs +\!\!+\ y,\ x \mapsto^1 v$
$= \mathsf{H} +\!\!+\ y +\!\!+\ xs,\ x \mapsto^1 v$ by I.H.
$= \mathsf{H} +\!\!+\ y +\!\!+\ x$ 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 Γ, $[\![\Gamma]\!]$ defines a dependency heap, with all x becoming $x \mapsto^n ()$ if x appears n times in Γ.

Lemma 6.
$[\![\Gamma,\ x]\!] \ \vdash\ x\ \dashv\ [\![\Gamma]\!]$. Similarly, if $[\![\Delta]\!] \ \vdash\ e\ \dashv\ \mathsf{H}$, then $[\![\Gamma, x]\!] \ \vdash\ \mathsf{drop}\ x;\ e\ \dashv\ \mathsf{H}$.

Proof. The goal holds by rule DRVAR (DRDROP) when $x \in \Gamma$ or by rule DRVARCON (DRDROPCON) if $x \notin \Gamma$. □

Lemma 7. (*linear resource calculus relates to reference counting*)
If $\Delta \mid \Gamma\ \vdash\ e\ \rightsquigarrow e'$, then $[\![\Delta, \Gamma]\!] \ \vdash\ e'\ \dashv\ [\![\Delta]\!]$.

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

$\Delta \mid x\ \vdash\ x\ \rightsquigarrow x$	given
$[\![\Delta,\ x]\!] \ \vdash\ x\ \dashv\ [\![\Delta]\!]$	Lemma 6

case

$\Delta \mid \Gamma\ \vdash\ e\ \rightsquigarrow \mathsf{dup}\ x;\ e'$	given
$\Delta \mid \Gamma, x\ \vdash\ e\ \rightsquigarrow e'$	given
$x\ \in \Delta, \Gamma$	given
$[\![\Delta, \Gamma, x]\!] \ \vdash\ e'\ \dashv\ [\![\Delta]\!]$	I.H.
$[\![\Delta, \Gamma]\!] \ \vdash\ \mathsf{dup}\ x;\ e'\ \dashv\ [\![\Delta]\!]$	DRDUP

case

$\Delta \mid \Gamma, x\ \vdash\ e\ \rightsquigarrow \mathsf{drop}\ x;\ e'$	given
$\Delta \mid \Gamma\ \vdash\ e\ \rightsquigarrow e'$	given
$[\![\Delta, \Gamma]\!] \ \vdash\ e'\ \dashv\ [\![\Delta]\!]$	I.H.
$[\![\Delta, \Gamma,\ x]\!] \ \vdash\ \mathsf{drop}\ x;\ e'\ \dashv\ [\![\Delta]\!]$	Lemma 6

case

$\Delta \mid \Gamma\ \vdash\ \lambda x.\ e\ \rightsquigarrow \lambda^{ys}\ x.\ e'$	given
$\varnothing \mid ys, x\ \vdash\ e\ \rightsquigarrow e'$	given
$ys\ =\ \mathsf{fv}(\lambda x.\ e)$	given
$[\![\Delta, ys]\!] \ \vdash\ ys\ \dashv\ [\![\Delta]\!]$	Lemma 6
$[\![ys, x]\!] \ \vdash\ e'\ \dashv\ \varnothing$	I.H.
$[\![\Delta,\ ys]\!] \ \vdash\ \lambda^{ys}\ x.\ e\ \dashv\ [\![\Delta]\!]$	DRLAM

case

$\Delta \mid \Gamma_1, \Gamma_2\ \vdash\ e_1\ e_2\ \rightsquigarrow e_1'\ e_2'$	given
$\Delta, \Gamma_2 \mid \Gamma_1\ \vdash\ e_1\ \rightsquigarrow e_1'$	given
$\Delta \mid \Gamma_2\ \vdash\ e_2\ \rightsquigarrow e_2'$	given
$[\![\Delta, \Gamma_1, \Gamma_2]\!] \ \vdash\ e_1'\ \dashv\ [\![\Delta,\ \Gamma_2]\!]$	I.H.
$[\![\Delta,\ \Gamma_2]\!] \ \vdash\ e_2'\ \dashv\ [\![\Delta]\!]$	I.H.
$[\![\Delta, \Gamma_1, \Gamma_2]\!] \ \vdash\ e_1'\ e_2'\ \dashv\ [\![\Delta]\!]$	DRAPP

case

$\Delta \mid \Gamma_1, \Gamma_2 \;\vdash\; \mathsf{val}\; x \;=\; e_1;\; e_2 \;\rightsquigarrow \mathsf{val}\; x \;=\; e_1';\; e_2'$	given
$\Delta, \Gamma_2 \mid \Gamma_1 \;\vdash\; e_1 \;\rightsquigarrow e_1'$	given
$\Delta \mid \Gamma_2, x \;\vdash\; e_2 \;\rightsquigarrow e_2'$	given
$x \notin \Delta,\; \Gamma_1,\; \Gamma_2$	given
$[\![\Delta, \Gamma_1, \Gamma_2]\!] \;\vdash\; e_1' \;\dashv\; [\![\Delta, \Gamma_2]\!]$	I.H.
$[\![\Delta,\; \Gamma_2,\; x]\!] \;\vdash\; e_2' \;\dashv\; [\![\Delta]\!]$	I.H.
$x \notin [\![\Delta]\!]$	follows
$[\![\Delta, \Gamma_1, \Gamma_2]\!] \;\vdash\; \mathsf{val}\; x \;=\; e_1';\; e_2' \;\dashv\; [\![\Delta]\!]$	DRBIND

case

$\Delta \mid \Gamma, x \;\vdash\; \mathsf{match}\; x\; \{\; \overline{p_i \;\mapsto e_i}\; \} \;\rightsquigarrow \mathsf{match}\; x\; \{\; \overline{p_i \;\mapsto e_i'}\; \}$	given
$\Delta \mid \Gamma, \mathsf{bv}(p_i) \;\vdash\; e_i \;\rightsquigarrow e_i'$	given
$[\![\Delta, \Gamma, x]\!] \;\vdash\; x \;\dashv\; [\![\Delta, \Gamma]\!]$	Lemma 6
$[\![\Delta,\; \Gamma,\; \mathsf{bv}(p_i)]\!] \;\vdash\; e_i' \;\dashv\; [\![\Delta]\!]$	I.H.
$[\![\Delta, \Gamma]\!] \;\vdash\; \mathsf{match}\; x\; \{\; \overline{p_i \;\mapsto e_i'}\; \} \;\dashv\; [\![\Delta]\!]$	DRMATCH

case

$\Delta \mid \Gamma_1, \ldots, \Gamma_n \;\vdash\; C\; v_1 \ldots v_n \;\rightsquigarrow C\; v_1' \ldots v_n'$	given
$\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \;\vdash\; v_i$	given
$[\![\Delta, \Gamma_i, \Gamma_{i+1}, \ldots, \Gamma_n]\!] \;\vdash\; v_i - \mid [\![\Delta, \Gamma_{i+1}, \ldots, \Gamma_n]\!]$	I.H.
$[\![\Delta, \Gamma_1, \ldots, \Gamma_n]\!] \;\vdash\; C\; v_1 \ldots v_n \;\dashv\; [\![\Delta]\!]$	DRCON

□

D.1.2 Weakening.

Lemma 8. (*Weakening*)
If $\mathsf{H}_1 \;\vdash\; e \;\dashv\; \mathsf{H}_2$, then $\mathsf{H}_1 \mathbin{+\!\!+} x \;\vdash\; e \;\dashv\; \mathsf{H}_2 \mathbin{+\!\!+} x$.

Proof. (*Of Lemma 8*) By induction on the judgment.
case

$\mathsf{H} \;\vdash\; C\; v_1 \ldots v_n - \mid \mathsf{H}_n$	given
$\mathsf{H} \;\vdash\; vi \;\dashv\; \mathsf{H}_i$	DRCON
$\mathsf{H} \mathbin{+\!\!+} x \;\vdash\; vi \;\dashv\; \mathsf{H}_i \mathbin{+\!\!+} x$	I.H.
$\mathsf{H} \mathbin{+\!\!+} x \;\vdash\; C\; v_1 \ldots v_n \;\dashv\; \mathsf{H}_n \mathbin{+\!\!+} x$	DRCON

case

$\mathsf{H} \;\vdash\; y \;\dashv\; \mathsf{H}_2$	given
$\mathsf{H} \;=\; \mathsf{H}_2 \mathbin{+\!\!+} y$	Lemma 3
$\mathsf{H} \mathbin{+\!\!+} x \;=\; \mathsf{H}_2 \mathbin{+\!\!+} y \mathbin{+\!\!+} x$	
$=\; \mathsf{H}_2 \mathbin{+\!\!+} x \mathbin{+\!\!+} y$	Lemma 5
$\mathsf{H}_2 \mathbin{+\!\!+} x \mathbin{+\!\!+} y \;\vdash\; y \;\dashv\; \mathsf{H}_2 \mathbin{+\!\!+} x$	Lemma 4

case

$\mathsf{H},\; y \mapsto^{n} ys \;\vdash\; \mathsf{dup}\; y;\; e \;\dashv\; \mathsf{H}_1$	given
$\mathsf{H},\; y \mapsto^{n+1} ys \;\vdash\; e \;\dashv\; \mathsf{H}_1$	DRDUP
$(\mathsf{H},\; y \mapsto^{n} ys) \mathbin{+\!\!+} y \;\vdash\; e \;\dashv\; \mathsf{H}_1$	definition of $\mathbin{+\!\!+}$
$(\mathsf{H},\; y \mapsto^{n} ys) \mathbin{+\!\!+} y \mathbin{+\!\!+} x \;\vdash\; e \;\dashv\; \mathsf{H}_1 \mathbin{+\!\!+} x$	I.H.
$(\mathsf{H},\; y \mapsto^{n} ys) \mathbin{+\!\!+} y \mathbin{+\!\!+} x$	
$=\; (\mathsf{H},\; y \mapsto^{n} ys) \mathbin{+\!\!+} x \mathbin{+\!\!+} y$	Lemma 5
$(\mathsf{H},\; y \mapsto^{n} ys) \mathbin{+\!\!+} x \mathbin{+\!\!+} y \;\vdash\; e \;\dashv\; \mathsf{H}_1 \mathbin{+\!\!+} x$	By substitution
$(\mathsf{H},\; y \mapsto^{n} ys) \mathbin{+\!\!+} x \;\vdash\; \mathsf{dup}\; y;\; e \;\dashv\; \mathsf{H}_1 \mathbin{+\!\!+} x$	DRDUP

case

$\mathsf{H} \vdash \mathsf{drop}\ y;\ e \dashv \mathsf{H}_2$	given
$\mathsf{H} \vdash \mathsf{drop}\ y;\ () \dashv \mathsf{H}_3$	follows
$\mathsf{H}_3 \vdash e \dashv \mathsf{H}_2$	above
$\mathsf{H} = \mathsf{H}_3 +\!\!\!+\, y$	Lemma 3
$\mathsf{H} +\!\!\!+\, x = \mathsf{H}_3 +\!\!\!+\, y +\!\!\!+\, x$	
$= \mathsf{H}_3 +\!\!\!+\, x +\!\!\!+\, y$	Lemma 5
$\mathsf{H}_3 +\!\!\!+\, x +\!\!\!+\, y \vdash \mathsf{drop}\ y;\ () \dashv \mathsf{H}_3 +\!\!\!+\, x$	Lemma 4
$\mathsf{H}_3 +\!\!\!+\, x \vdash e \dashv \mathsf{H}_2 +\!\!\!+\, x$	I.H.
$\mathsf{H}_3 +\!\!\!+\, x +\!\!\!+\, y \vdash \mathsf{drop}\ y;\ e \dashv \mathsf{H}_2 +\!\!\!+\, x$	Follows
$\mathsf{H} +\!\!\!+\, x \vdash \mathsf{drop}\ y;\ e \dashv \mathsf{H}_2 +\!\!\!+\, x$	By substitution

case

$\mathsf{H} \vdash \lambda^{ys}\, z.\ e \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash ys \dashv \mathsf{H}_1$	given
$ys \mapsto^1 (),\ z \mapsto^1 () \vdash e \dashv \varnothing$	given
$\mathsf{H} +\!\!\!+\, x \vdash ys \dashv \mathsf{H}_1 +\!\!\!+\, x$	I.H.
$\mathsf{H} +\!\!\!+\, x \vdash \lambda z.\ e \dashv \mathsf{H}_1 +\!\!\!+\, x$	DRLAM

case

$\mathsf{H} \vdash e_1\ e_2 \dashv \mathsf{H}_2$	given
$\mathsf{H} \vdash e_1 \dashv \mathsf{H}_1$	given
$\mathsf{H}_1 \vdash e_2 \dashv \mathsf{H}_2$	given
$\mathsf{H} +\!\!\!+\, x \vdash e_1 \dashv \mathsf{H}_1 +\!\!\!+\, x$	I.H.
$\mathsf{H}_1 +\!\!\!+\, x \vdash e_2 \dashv \mathsf{H}_2 +\!\!\!+\, x$	I.H.
$\mathsf{H} +\!\!\!+\, x \vdash e_1\ e_2 \dashv \mathsf{H}_2 +\!\!\!+\, x$	DRAPP

case

$\mathsf{H} \vdash \mathsf{val}\ z = e_1\ ;\ e_2 \dashv \mathsf{H}_2$	given
$\mathsf{H} \vdash e_1 \dashv \mathsf{H}_1$	given
$\mathsf{H}_1, z \mapsto^1 () \vdash e_2 \dashv \mathsf{H}_2$	given
$z \notin \mathsf{H}$	given
$\mathsf{H} +\!\!\!+\, x \vdash e_1 \dashv \mathsf{H}_1 +\!\!\!+\, x$	I.H.
$(\mathsf{H}_1,\ z \mapsto^1 ()) +\!\!\!+\, x \vdash e_2 \dashv \mathsf{H}_2 +\!\!\!+\, x$	I.H.
$z \notin \mathsf{H}_1$	Lemma 2
$\mathsf{H}_1, z \mapsto^1 () = \mathsf{H}_1 +\!\!\!+\, z$	
$(\mathsf{H}_1,\ z \mapsto^1 ()) +\!\!\!+\, x = \mathsf{H}_1 +\!\!\!+\, z +\!\!\!+\, x$	
$= \mathsf{H}_1 +\!\!\!+\, x +\!\!\!+\, z$	Lemma 5
$= \mathsf{H}_1 +\!\!\!+\, x,\ z \mapsto^1 ()$	
$\mathsf{H}_1 +\!\!\!+\, x,\ z \mapsto^1 () \vdash e_2 \dashv \mathsf{H}_2 +\!\!\!+\, x$	by substitution
$\mathsf{H} +\!\!\!+\, x \vdash e_1\ e_2 \dashv \mathsf{H}_2 +\!\!\!+\, x$	DRBIND

case

$\mathsf{H} \vdash \mathsf{match}\ z\ \{\ \overline{p_i \mapsto e_i}\ \} \dashv \mathsf{H}'$	given
$\mathsf{H} \vdash z \dashv \mathsf{H}_1$	given
$\mathsf{H}_1, \mathsf{H}_i \vdash e_i \dashv \mathsf{H}'$	given
$\mathsf{H}_i = [\![\mathsf{bv}(p_i)]\!]$	given
$\mathsf{H} \mathbin{+\!\!+} x \vdash z \dashv \mathsf{H}_1 \mathbin{+\!\!+} x$	I.H.
$(\mathsf{H}_1, \mathsf{H}_i) \mathbin{+\!\!+} x \vdash e_i \dashv \mathsf{H}' \mathbin{+\!\!+} x$	I.H.
$\mathsf{bv}(p_i)$ fresh	assume
$\mathsf{H}_1,\ [\![\mathsf{H}_i]\!] = \mathsf{H}_1 \mathbin{+\!\!+} \mathsf{bv}(p_i)$	
$(\mathsf{H}_1,\ \mathsf{H}_i) \mathbin{+\!\!+} x = \mathsf{H}_1 \mathbin{+\!\!+} \mathsf{bv}(p_i) \mathbin{+\!\!+} x$	
$= \mathsf{H}_1 \mathbin{+\!\!+} x \mathbin{+\!\!+} \mathsf{bv}(p_i)$	Lemma 5
$= \mathsf{H}_1 \mathbin{+\!\!+} x,\ \mathsf{H}_i$	
$\mathsf{H}_1 \mathbin{+\!\!+} x, \mathsf{H}_i \vdash e_i \dashv \mathsf{H}' \mathbin{+\!\!+} x$	by substitution
$\mathsf{H} \mathbin{+\!\!+} x \vdash \mathsf{match}\ z\ \{\ \overline{p_i \mapsto e_i}\ \} \dashv \mathsf{H}' \mathbin{+\!\!+} x$	DRMATCH

□

D.2 Soundness of Reference Counting Semantics

Lemma 9. (*Reference counting semantics is sound (small step)*)
If $e_1 \longrightarrow e_2$, and H_1 ok, and e_1' ok, and $[\mathsf{H}_1]e_1' = e_1$, and $\mathsf{H}_1 \vdash e_1' \dashv \mathsf{H}_1'$, then there exists H_2, e_2' such that $\mathsf{H}_1 \mid e_1' \longmapsto^*{}_r \mathsf{H}_2 \mid e_2'$, and $[\mathsf{H}_2]e_2' = e_2$.

Proof. (*Of Lemma 9*) By induction on the evaluation judgment.

case $(app)\ (\lambda x.\ e)\ v \longrightarrow e[x{:=}v]$

$[\mathsf{H}_1]e_1' = (\lambda x.\ e)\ v$ given

subcase $e_1' = f\ z$.

$[\mathsf{H}_1](f\ z) = (\lambda x.\ e)\ v$	given
$[\mathsf{H}_1]f = (\lambda x.\ e)$	follows
$[\mathsf{H}_1]z = v$	follows
$f \mapsto^n \lambda^{ys}\ x.\ e' \in \mathsf{H}_1$	follows
$[\mathsf{H}_1]e' = e$	above
$\mathsf{H}_1 \mid f\ z \longrightarrow_r \mathsf{H}_1 \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ f;\ e'[x{:=}z]$	(app)
$\mathsf{H}_1 \vdash f\ z \dashv \mathsf{H}_1'$	given
$\mathsf{H}_1 = \mathsf{H}_1' \mathbin{+\!\!+} f \mathbin{+\!\!+} z$	Lemma 3
H_1 ok	given
$ys \in \mathsf{H}_1$	$f \mapsto^n \lambda^{ys}\ x \cdot e' \in \mathsf{H}_1$
$\mathsf{H}_1 \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ f;\ e'[x{:=}z]$	
$\longrightarrow_r \mathsf{H}_1 \mathbin{+\!\!+} ys \mid \mathsf{drop}\ f;\ e'[x{:=}z]$	(dup) and $\mathbin{+\!\!+}$
$\mathsf{H}_1 \mathbin{+\!\!+} ys = \mathsf{H}_1' \mathbin{+\!\!+} f \mathbin{+\!\!+} z \mathbin{+\!\!+} ys$	
$= \mathsf{H}_1' \mathbin{+\!\!+} ys \mathbin{+\!\!+} z \mathbin{+\!\!+} f$	Lemma 5
$\mathsf{H}_1 \mathbin{+\!\!+} ys \mid \mathsf{drop}\ f;\ e'[x{:=}z]$	
$\longrightarrow_r \mathsf{H}_1' \mathbin{+\!\!+} ys \mathbin{+\!\!+} z \mid e'[x{:=}z]$	$(drop)$ and Lemma 4
$\mathsf{H}_1' \mathbin{+\!\!+} ys \mathbin{+\!\!+} z \vdash e'[x{:=}z] \dashv \mathsf{H}_1'$	Lemma 14
$\mathsf{fv}(e'[x{:=}z]) \in \mathsf{H}_1' \mathbin{+\!\!+} ys \mathbin{+\!\!+} z$	Lemma 2
$\mathsf{H}_1 = \mathsf{H}_1' \mathbin{+\!\!+} f \mathbin{+\!\!+} z$	known
$ys \in \mathsf{H}_1$	known
$\mathsf{fv}(e'[x{:=}z]) \in \mathsf{H}_1$	follows
$[\mathsf{H}_1' \mathbin{+\!\!+} ys \mathbin{+\!\!+} z](e'[x{:=}z])$	
$= [\mathsf{H}_1](e'[x{:=}z])$	follows
$= ([\mathsf{H}_1]e')[x{:=}[\mathsf{H}_1]z]$	by substitution
$= e[x{:=}v]$	

subcase $e_1' = (\lambda x.\ e')\ z$.

$[\mathsf{H}_1]((\lambda x.\ e')\ z)\ =\ (\lambda x.\ e)\ v$	given
$[\mathsf{H}_1]e'\ =\ e$	follows
$[\mathsf{H}_1]z\ =\ v$	follows
$\mathsf{H} \mid (\lambda x.\ e')\ z \longmapsto \mathsf{H},\ f \mapsto^1 \lambda^{ys} x.\ e' \mid f\ z$	(lam) and step
f fresh, $ys\ =\ \mathsf{fv}(\lambda x.\ e')$	above
$\mathsf{H}, f \mapsto^1 \lambda^{ys} x.e'\ \vdash\ f\ z\ \dashv\ \mathsf{H}_1'$	Lemma 15
$\mathsf{H}, f \mapsto^1 \lambda^{ys} x.e'$ ok	above
$f\ z$ ok	above
$[\mathsf{H}, f \mapsto^1 \lambda^{ys} x.e'](f\ z)\ =\ (\lambda x.\ e\ v)$	by substitution
follows by the previous subcase	

subcase The rest subcases are $e_1'\ =\ f\ v'$ and $e_1'\ =\ (\lambda x.\ e')\ v'$ 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 $(C\ v_1 \ldots v_n)\ \{\overline{p_i \to e_i}\} \longrightarrow e_i[x_1{:=}v_1, \ldots, x_n{:=}v_n]$ with $p_i\ =\ C\ x_1 \ldots x_n$.

$[\mathsf{H}_1]e_1'\ =\ \mathsf{match}\ (C\ v_1 \ldots v_n)\ \{\overline{p_i \to e_i}\}$	given
$e_1'\ =\ \mathsf{match}\ x\ \{\overline{p_i \to e_i'}\}$	follows
$[\mathsf{H}_1]x\ =\ C\ v_1 \ldots v_n$	above
$[\mathsf{H}_1]e_i'\ =\ e_i$	above
$x \mapsto^n C\ y_1 \ldots y_n\ \in \mathsf{H}_1$	follows
$[\mathsf{H}_1]y_1\ =\ v_1,\ \ldots,\ [\mathsf{H}_1]y_n\ =\ v_n$	above
$\mathsf{H}_1 \mid \mathsf{match}\ x\ \{\overline{p_i \to e_i'}\}$	
$\longrightarrow_r \mathsf{H}_1\ \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ x;\ e_i'[xs{:=}ys]$	$(match)$
$p_i\ =\ C\ xs$	above
$\mathsf{H}_1\ \vdash\ \mathsf{match}\ x\ \{\overline{p_i \to e_i'}\}\ \dashv\ \mathsf{H}_1'$	given
$\mathsf{H}_1\ \vdash\ x\ \dashv\ \mathsf{H}_2$	DRMATCH
$\mathsf{H}_2,\ [\![xs]\!]\ \vdash\ e_i\ \dashv\ \mathsf{H}'$	above
$\mathsf{H}_1\ =\ \mathsf{H}_2 +\!\!+\ x$	Lemma 3
$\mathsf{H}_1 \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ x;\ e_i[xs{:=}ys]$	
$\longrightarrow_r \mathsf{H}_1 +\!\!+\ ys\ \mid \mathsf{drop}\ x;\ e_i[xs{:=}ys]$	(dup) and $+\!\!+$
$\mathsf{H}_1 +\!\!+\ ys\ =\ \mathsf{H}_2 +\!\!+\ x +\!\!+\ ys$	
$= \mathsf{H}_2 +\!\!+\ ys +\!\!+\ x$	Lemma 5
$\mathsf{H}_1 +\!\!+\ ys \mid \mathsf{drop}\ x;\ e_i[xs{:=}ys] \longrightarrow_r \mathsf{H}_2{+\!\!+}ys \mid e_i[xs{:=}ys]$	$(drop)$ and Lemma 4
$\mathsf{H}_1\ =\ \mathsf{H}_2 +\!\!+\ x$	known
H_1 and $\mathsf{H}_2 +\!\!+\ ys$ differs only in x	
$\mathsf{H}_2 +\!\!+\ ys\ \vdash\ e_i[xs{:=}ys]\ \dashv\ \mathsf{H}_1'$	Lemma 15
$\mathsf{fv}(e_i[xs{:=}ys])\ \in \mathsf{H}_2{+\!\!+}ys$	Lemma 2
$[\mathsf{H}_2{+\!\!+}ys](e_i[xs{:=}ys]) = [\mathsf{H}_1](e_i[xs{:=}ys])$	
$([\mathsf{H}_1]e_i)(xs{:=}[\mathsf{H}_1]ys)$	by substitution
$(e_i)(xs{:=}vs)$	

case (dup) dup $x;\ e \longrightarrow e$

$[\mathsf{H}_1]e_1'\ =\ \mathsf{dup}\ x;\ e$	given
$e_1'\ =\ \mathsf{dup}\ x;\ e'$	follows
$[\mathsf{H}_1]e'\ =\ e$	above
$\mathsf{H}_1\ \vdash\ \mathsf{dup}\ x;\ e'\ \dashv\ \mathsf{H}_1'$	given
$x\ \in \mathsf{H}_1$	follows
$\mathsf{H}_1 \mid \mathsf{dup}\ x;\ e' \longrightarrow_r \mathsf{H}_1 +\!\!+\ x \mid e'$	(dup) and $+\!\!+$
$[\mathsf{H}_1 +\!\!+\ x]e'\ =\ [\mathsf{H}_1]e'\ =\ e$	

case $(drop)$ drop $x;\ e \longrightarrow e$

$[\mathsf{H}_1]e_1' = \mathsf{drop}\ x;\ e$	given
$e_1' = \mathsf{drop}\ x;\ e'$	follows
$[\mathsf{H}_1]e' = e$	above
$\mathsf{H}_1 \vdash \mathsf{drop}\ x;\ e' \dashv \mathsf{H}_1'$	given
$\mathsf{H}_1 \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}_2$	follows
$\mathsf{H}_2 \vdash e' \dashv \mathsf{H}_1'$	above
$\mathsf{H}_1 = \mathsf{H}_2 +\!\!\!+\ x$	Lemma 3
$\mathsf{H}_1 \mid \mathsf{drop}\ x;\ e' \longrightarrow_r \mathsf{H}_2 \mid e'$	$(drop)$ and Lemma 4
$\mathsf{H}_2 \vdash e' \dashv \mathsf{H}_1'$	Lemma 15
$\mathsf{fv}(e') \in \mathsf{H}_2$	Lemma 2
$[\mathsf{H}_2]e' = [\mathsf{H}_1]e' = e$	

□

Lemma 10. (*Reference counting semantics is sound (big step, part 1)*)
Given $\mathsf{E}[e_1]$, and H_1 ok, and e_1' ok, and $[\mathsf{H}_1]e_1' = \mathsf{E}[e_1]$, and $\mathsf{H}_1 \vdash e_1' \dashv \mathsf{H}_1'$, then there exists H_2, E'' and e_2' such that $\mathsf{H}_1 \mid e_1' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}'[e_2']$, and $[\mathsf{H}_2]\mathsf{E}' = \mathsf{E}$, and $[\mathsf{H}_2]e_2' = e_1$.

Proof. (*Of Lemma 10*) By induction on the evaluation context E.
case $\mathsf{E} = \square$. Let $\mathsf{E}' = \square$ and $e_2' = e_1'$, then the goals follow trivially.
case $\mathsf{E} = \mathsf{E}_1\ e$.

$[\mathsf{H}_1]e_1' = \mathsf{E}_1[e_1]\ e$	given
$e_1' = e_2'\ e_3'$	for some e_2', e_3'
$[\mathsf{H}_1]e_2' = \mathsf{E}_1[e_1]$	follows
$[\mathsf{H}_1]e_3' = e$	follows
$\mathsf{H}_1 \vdash e_2'\ e_3' \dashv \mathsf{H}_1'$	given
$\mathsf{H}_1 \vdash e_2' \dashv \mathsf{H}_3$	DRAPP
$\mathsf{H}_3 \vdash e_3' \dashv \mathsf{H}_1'$	above
$\mathsf{H}_1 \vdash e_2' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}'[e_2']$	I.H.
$[\mathsf{H}_2]\mathsf{E}' = \mathsf{E}_1$	above
$[\mathsf{H}_2]e_2' = e_1$	above
$\mathsf{E}'' = \mathsf{E}'\ e_3'$	let
$\mathsf{H}_1 \vdash e_2'\ e_3' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}'[e_2']\ e_3'$	step
$\mathsf{H}_1 \vdash e_1' \longrightarrow^*{}_r \mathsf{H}_2\ \mid \mathsf{E}''[e_2']$	by substitution
$\mathsf{H}_2 \vdash \mathsf{E}'[e_2']\ e_3' \dashv \mathsf{H}_1'$	Lemma 15
$\mathsf{fv}(e_3') \in \mathsf{H}_2$	Lemma 2
$[\mathsf{H}_2]\mathsf{E}'' = [\mathsf{H}_2]\mathsf{E}'\ [\mathsf{H}_2]e_3'$	
$= [\mathsf{H}_2]\mathsf{E}'\ [\mathsf{H}_1]e_3'$	
$= \mathsf{E}_1\ e$	

case $\mathsf{E} = x\ \mathsf{E}_1$.

$[\mathsf{H}_1]e_1' = x\ \mathsf{E}_1[e_1]$	given
$e_1' = y\ e_3'$	for some y, e_3'
$[\mathsf{H}_1]y = x$	follows
$[\mathsf{H}_1]e_3' = \mathsf{E}_1[e_1]$	follows
$\mathsf{H}_1 \vdash y\ e_3' \dashv \mathsf{H}_1'$	given
$\mathsf{H}_1 \vdash y \dashv \mathsf{H}_3$	DRAPP
$\mathsf{H}_3 \vdash e_3' \dashv \mathsf{H}_1'$	above
$\mathsf{H}_1 = \mathsf{H}_3 +\!\!+\ y$	Lemma 3
$\mathsf{H}_1 \vdash e_3' \dashv \mathsf{H}_1' +\!\!+\ y$	Lemma 8
$\mathsf{H}_1 \vdash e_3' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}'[e_2']$	I.H.
$[\mathsf{H}_2]\mathsf{E}' = \mathsf{E}_1$	above
$[\mathsf{H}_2]e_2' = e_1$	above
$\mathsf{E}'' = y\ \mathsf{E}'$	let
$\mathsf{H}_1 \vdash y\ e_3' \longrightarrow^*{}_r \mathsf{H}_2 \mid y\ \mathsf{E}'[e_2']$	step
$\mathsf{H}_1 \vdash e_1' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}''[e_2']$	by substitution
$\mathsf{H}_2 \vdash y\ \mathsf{E}'[e_2'] \dashv \mathsf{H}_1'$	Lemma 15
$\mathsf{y} \in \mathsf{H}_2$	Lemma 2
$[\mathsf{H}_2]\mathsf{E}'' = [\mathsf{H}_2]y\ [\mathsf{H}_2]\mathsf{E}'$	
$= [\mathsf{H}_1]y\ [\mathsf{H}_2]\mathsf{E}'$	
$= x\ \mathsf{E}_1$	

case $\mathsf{E} = v\ \mathsf{E}_1$ where v is not a variable.

$[\mathsf{H}_1]e_1' = v\ \mathsf{E}_1[e_1]$	given
$e_1' = v'\ e_3'$	for some v', e_3'
$[\mathsf{H}_1]v' = v$	follows
$[\mathsf{H}_1]e_3' = \mathsf{E}_1[e_1]$	follows
$\mathsf{H}_1 \vdash v'\ e_3' \dashv \mathsf{H}_1'$	given
$\mathsf{H}_1 \vdash v' \dashv \mathsf{H}_3$	DRAPP
$\mathsf{H}_3 \vdash e_3' \dashv \mathsf{H}_1'$	above
$\mathsf{H}_1 = \mathsf{H}_3 +\!\!+\ \mathsf{fv}(v')$	Lemma 3
$\mathsf{H}_1 \vdash v'\ e_3' \longrightarrow \mathsf{H}_1,\ z \mapsto^1 v' \mid z\ e_3'$	(lam) or (con)
$\mathsf{H}_1,\ z \mapsto^1 v' \vdash e_3' \dashv \mathsf{H}_1' +\!\!+\ \mathsf{fv}(v'), z \mapsto^1 v'$	Lemma 8
$\mathsf{H}_1, z \mapsto^1 v' \vdash e_3' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}'[e_2']$	I.H.
$[\mathsf{H}_2]\mathsf{E}' = \mathsf{E}_1$	above
$[\mathsf{H}_2]e_2' = e_1$	above
$\mathsf{E}'' = z\ \mathsf{E}'$	let
$\mathsf{H}_1 \vdash v'\ e_3' \longrightarrow^*{}_r \mathsf{H}_2 \mid z\ \mathsf{E}'[e_2']$	step
$\mathsf{H}_1 \vdash e_1' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}''[e_2']$	by substitution
$\mathsf{H}_2 \vdash v'\ \mathsf{E}'[e_2'] \dashv \mathsf{H}_1'$	Lemma 15
$\mathsf{fv}(v') \in \mathsf{H}_2$	Lemma 2
$[\mathsf{H}_2]\mathsf{E}'' = [\mathsf{H}_2]v'\ [\mathsf{H}_2]\mathsf{E}'$	
$= [\mathsf{H}_1]v'\ [\mathsf{H}_2]\mathsf{E}'$	
$= v\ \mathsf{E}_1$	

case $\mathsf{E} = \mathsf{val}\ x = \mathsf{E}_1;\ e.$

$[\mathsf{H}_1]e_1' = \mathsf{val}\ x = \mathsf{E}_1[e_1];\ e$	given
$e_1' = \mathsf{val}\ x = e_2';\ e_3'$	for some e_2', e_3'
$[\mathsf{H}_1]e_2' = \mathsf{E}_1[e_1]$	follows
$[\mathsf{H}_1]e_3' = e$	follows
$\mathsf{H}_1 \vdash \mathsf{val}\ x = \mathsf{E}_1[e_1];\ e \dashv \mathsf{H}_1'$	given
$\mathsf{H}_1 \vdash e_2' \dashv \mathsf{H}_3$	drbind
$\mathsf{H}_1 \vdash e_2' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}'[e_2']$	I.H.
$[\mathsf{H}_2]\mathsf{E}' = \mathsf{E}_1$	above
$[\mathsf{H}_2]e_2' = e_1$	above
$\mathsf{E}'' = \mathsf{val}\ x = \mathsf{E}';\ e_3'$	let
$\mathsf{H}_1 \vdash \mathsf{val}\ x = e_2';\ e_3' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{val}\ x = \mathsf{E}'[e_2'];\ e_3'$	step
$\mathsf{H}_1 \vdash e_1' \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}''[e_2']$	by substitution
$\mathsf{H}_2 \vdash \mathsf{val}\ x = \mathsf{E}'[e_2'];\ e_3' \dashv \mathsf{H}_1'$	Lemma 15
$\mathsf{fv}(e_3') \in \mathsf{H}_2, [\![x]\!]$	Lemma 2
$[\mathsf{H}_2]\mathsf{E}'' = \mathsf{val}\ x = [\mathsf{H}_2]\mathsf{E}';\ [\mathsf{H}_2]e_3'$	
$= [\mathsf{H}_2]\mathsf{E}'\ [\mathsf{H}_1]e_3'$	
$= \mathsf{E}_1\ e$	

□

Lemma 11. (*Reference counting semantics is sound (big step, part 2)*)
If $\mathsf{E}[e_1] \longmapsto \mathsf{E}[e_2]$, and H_1 ok, and $\mathsf{E}'[e_1']$ ok, and $[\mathsf{H}_1]\mathsf{E}' = \mathsf{E}$, and $[\mathsf{H}_1]e_1' = e_1$, and $\mathsf{H}_1 \vdash \mathsf{E}[e_1'] \dashv \mathsf{H}_1'$, then there exists H_2, e_2' such that $\mathsf{H}_1 \mid \mathsf{E}'[e_1'] \longrightarrow^*{}_r \mathsf{H}_2 \mid \mathsf{E}'[e_2']$, and $[\mathsf{H}_2]\mathsf{E}' = \mathsf{E}$. and $[\mathsf{H}_2]e_2' = e_2$.

Proof. (*Of Lemma 11*) By induction on the evaluation context E. Note that following Lemma 15 and 2, we have $\mathsf{fv}(\mathsf{E}') \in \mathsf{H}_2$. So $[\mathsf{H}_2]\mathsf{E}' = [\mathsf{H}_1]\mathsf{E}' = \mathsf{E}$.

case $\mathsf{E} = \square$. Then $\mathsf{E}' = \square$. The goal follows by Lemma 9.

case $\mathsf{E} = \mathsf{E}_1\ e$, then $\mathsf{E}' = \mathsf{E}_1'\ e'$.

$[\mathsf{H}_1]\mathsf{E}' = \mathsf{E}_1\ e$	given
$[\mathsf{H}_1]\mathsf{E}_1' = \mathsf{E}'$	above
$[\mathsf{H}_1]e' = e$	above
$\mathsf{E}_1[e_1] \longmapsto \mathsf{E}_1[e_2]$	given
$\mathsf{H}_1 \vdash \mathsf{E}_1'[e_1']\ e' \dashv \mathsf{H}_1'$	given
$\mathsf{H}_1 \vdash \mathsf{E}_1'[e_1'] \dashv \mathsf{H}_2$	drapp
$\mathsf{H}_2 \vdash e' \dashv \mathsf{H}_1'$	above
$\mathsf{H}_1 \mid \mathsf{E}_1'[e_1'] \longrightarrow^*{}_r \mathsf{H}_3 \mid \mathsf{E}_1'[e_2']$	I.H.
$[\mathsf{H}_3]\ e_2' = e_2$	above
$\mathsf{H}_1 \mid \mathsf{E}_1'[e_1']\ e' \longrightarrow_r \mathsf{H}_3 \mid \mathsf{E}_1'[e_2']\ e'$	$\longmapsto$

case $\mathsf{E} = v\ \mathsf{E}_1$, then $\mathsf{E}' = x\ \mathsf{E}_1'$.

$[\mathsf{H}_1]\mathsf{E}' = v\ \mathsf{E}_1$	given
$[\mathsf{H}_1]x' = v$	above
$[\mathsf{H}_1]\mathsf{E}_1' = \mathsf{E}_1$	above
$\mathsf{E}_1[e_1] \longmapsto \mathsf{E}_1[e_2]$	given
$\mathsf{H}_1 \vdash x\ \mathsf{E}_1'[e_1'] \dashv \mathsf{H}_1'$	given
$\mathsf{H}_1 \vdash x \dashv \mathsf{H}_2$	drapp
$\mathsf{H}_2 \vdash \mathsf{E}_1'[e_1'] \dashv \mathsf{H}_1'$	above
$\mathsf{H}_1 = \mathsf{H}_2 +\!\!+\ x$	Lemma 3
$\mathsf{H}_1 \vdash \mathsf{E}_1'[e_1'] \dashv \mathsf{H}_1' +\!\!+\ x$	Lemma 8
$\mathsf{H}_1 \mid \mathsf{E}_1'[e_1'] \longrightarrow_r \mathsf{H}_3 \mid \mathsf{E}_1'[e_2']$	I.H.
$[\mathsf{H}_3]\ e_2' = e_2$	above
$\mathsf{H}_1 \mid x\ \mathsf{E}_1'[e_1'] \longrightarrow_r \mathsf{H}_3 \mid x\ \mathsf{E}_1'[e_2']$	$\longmapsto$

case $\mathsf{E} = \mathsf{val}\ x = \mathsf{E}_1;\ e$, then $\mathsf{E}' = \mathsf{val}\ x = \mathsf{E}_1';\ e'$.

$[\mathsf{H}_1]\mathsf{E}' \;=\; \mathsf{val}\; x \;=\; \mathsf{E}_1;\; e$	given
$[\mathsf{H}_1]\mathsf{E}'_1 \;=\; \mathsf{E}'$	above
$[\mathsf{H}_1]e' \;=\; e$	above
$\mathsf{E}_1[e_1] \longmapsto \mathsf{E}_1[e_2]$	given
$\mathsf{H}_1 \;\vdash\; \mathsf{val}\; x \;=\; \mathsf{E}'_1[e'_1];\; e' \;\dashv\; \mathsf{H}'_1$	given
$\mathsf{H}_1 \;\vdash\; \mathsf{E}'_1[e'_1] \;\dashv\; \mathsf{H}_2$	DRBIND
$\mathsf{H}_1 \mid \mathsf{E}'_1[e'_1] \longrightarrow^*_r \mathsf{H}_3 \mid \mathsf{E}'_1[e'_2]$	I.H.
$[\mathsf{H}_3]\; e'_2 \;=\; e_2$	above
$\mathsf{H}_1 \mid \mathsf{val}\; x \;=\; \mathsf{E}'_1[e'_1];\; e' \longrightarrow_r \mathsf{H}_3 \mid \mathsf{val}\; x \;=\; \mathsf{E}'_1[e'_2];\; e'$	$\longmapsto$

□

Lemma 12. (*Reference counting semantics is sound (big step)*)
If $e_1 \longmapsto e_2$, and H_1 ok, and e'_1 ok, and $[\mathsf{H}_1]e'_1 \;=\; e_1$, and $\mathsf{H}_1 \;\vdash\; e'_1 \;\dashv\; \mathsf{H}'_1$, then there exists H_2 and e'_2 such that $\mathsf{H}_1 \mid e'_1 \longrightarrow^*_r \mathsf{H}_2 \mid e'_2$, and $[\mathsf{H}_2]e'_2 \;=\; e_2$.

Proof. (*Of Lemma 12*)

$e_1 \longmapsto e_2$	given
$e_1 \;=\; \mathsf{E}[e_3]$	suppose
$e_2 \;=\; \mathsf{E}[e_4]$	suppose
$\mathsf{E}[e_3] \longmapsto \mathsf{E}[e_4]$	given
$e'_1 \;=\; \mathsf{E}_1[e'_2]$	suppose
$\mathsf{H}_1 \mid \mathsf{E}_1[e'_2] \longrightarrow^*_r \mathsf{H}_2 \mid \mathsf{E}_2[e'_3]$	Lemma 10
$[\mathsf{H}_2]\mathsf{E}_2 \;=\; \mathsf{E}$	above
$[\mathsf{H}_2]e'_3 \;=\; e_3$	above
H_2 ok	Lemma 15
$\mathsf{E}_2[e'_3]$ ok	above
$\mathsf{H}_2 \;\vdash\; \mathsf{E}_2[e'_3] \;\dashv\; \mathsf{H}'_1$	above
$\mathsf{H}_2 \mid \mathsf{E}_2[e'_3] \longrightarrow \mathsf{H}_3 \mid \mathsf{E}_2[e'_4]$	Lemma 11
$[\mathsf{H}_3]\mathsf{E}_2 \;=\; \mathsf{E}$	above
$[\mathsf{H}_3]e'_4 \;=\; e_4$	above
$[\mathsf{H}_3](\mathsf{E}_2[e'_4]) \;=\; ([\mathsf{H}_3]\mathsf{E}_2)[\;[\mathsf{H}_3]e'_4\;]$	by substitution
$=\; \mathsf{E}[e_4] \;=\; e_2$	

□

Proof. (*Of Theorem 1*)

$\varnothing \mid \varnothing \;\vdash\; e \;\rightsquigarrow e'$	given
$e \longmapsto^* v$	given
$e' \longmapsto^* v$	Theorem 5
H ok	
e' ok	from LAM
$\varnothing \;\vdash\; e' \;\dashv\; \varnothing$	Lemma 7
$e' \longmapsto^*_r \mathsf{H}_2 \mid v'$	Lemma 12
$[\mathsf{H}_2]v' \;=\; v$	above

case $v' \;=\; x$. Then the goal is proved.
case v' is not a variable. Then by (lam) or (con) we have $\mathsf{H}_2 \mid v' \longrightarrow_r \mathsf{H}_2,\; z \mapsto^1 v' \mid z$, with z fresh. Then $[\mathsf{H}_2, z \mapsto^1 v']z \;=\; [\mathsf{H}_2, z \mapsto^1 v']v' \;=\; [\mathsf{H}_2]v' \;=\; v$. □

D.3 No Garbage

D.3.1 Extending strict evaluation semantics. If we add dup e and drop e in the syntax, as well as add to the standard semantics in Figure 6 the following rules:

$$(dup)\quad \mathsf{dup}\; e';\; e \;\longrightarrow\; e$$
$$(drop)\quad \mathsf{drop}\; e';\; e \;\longrightarrow\; e$$

we immediately see that translations does not change evaluation:

Theorem 5. (*Translation is sound*)
If $e \longmapsto^* v$ with $\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$, then also $e' \longmapsto^* v$.

Proof. (*Of Theorem 5*) Follows directly from Lemma 13 and the two reduction rule (dup) and $(drop)$. □

Lemma 13. (*Translation only inserts dup/drop*)
If $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$ then $e = \lceil e' \rceil$.

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

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 $[\![ys,\ x]\!] \vdash e_1 \dashv \varnothing$.

Definition 5. (*Well-formed Heap*)
If H ok, then (1) if $x \mapsto^n v \in \mathsf{H}$, then $\mathsf{fv}(v) \in \mathsf{H}$, 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 $\mathsf{H}_1 \vdash e_1 \dashv \mathsf{H}'$, and $\mathsf{H}_1 \mid e_1 \longrightarrow_r \mathsf{H}_2 \mid e_2$, then H_2 ok, e_2 ok, and $\mathsf{H}_2 \vdash e_2 \dashv \mathsf{H}'$.

Proof. (*Of Lemma 14*) When we a new variable $z \mapsto^1 v$ in the heap (e.g., (lam)), z is fresh so its domain cannot refer to z (even indirectly). So there is no dependency cycle. Also, in those cases, since $\mathsf{H}_1 \vdash v \dashv \mathsf{H}'$, by Lemma 2, we know $\mathsf{fv}(v) \in \mathsf{H}_1$. Moreover we have v ok as a precondition.

Heap reduction retains abstractions, with the only change being substitution. If $[\![ys,\ x]\!] \vdash e \dashv \varnothing$, then $[\![ys[y{:=}z],\ x]\!] \vdash e[y{:=}z] \dashv \varnothing$ by substitution.

Now we prove $\mathsf{H}_2 \vdash e_2 \dashv \mathsf{H}'$ by induction on the judgment.

case (app_r) $\mathsf{H} \mid f\ z \longrightarrow_r \mathsf{H} \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ f;\ e[x{:=}z]$ $\quad (f \mapsto^n \lambda^{ys} x.\ e) \in \mathsf{H}$

$\mathsf{H} \vdash f\ z \dashv \mathsf{H}_1$	given
$\mathsf{H} = \mathsf{H}_1 + f + z$	Lemma 3
$ys \in \mathsf{H}_1 + f + z$	$f \mapsto \lambda^{ys} x.\ e$
$\mathsf{H} \vdash \mathsf{dup}\ ys;\ () \dashv \mathsf{H} + ys$	by definition
$\mathsf{H} + ys = \mathsf{H}_1 + f + z + ys$	
$= \mathsf{H}_1 + ys + z + f$	Lemma 5
$\mathsf{H}_1 + ys + z + f \vdash \mathsf{drop}\ f;\ () \dashv \mathsf{H}_1 + ys + z$	Lemma 4
$[\![ys,\ x]\!] \vdash e \dashv \varnothing$	$\lambda^{ys} x.\ e$ ok
$[\![ys,\ z]\!] \vdash e[x{:=}z] \dashv \varnothing$	by substitution
$\mathsf{H}_1 + ys + z \vdash e[x{:=}z] \dashv \mathsf{H}_1$	Lemma 8

case $(match_r)$ $\mathsf{H} \mid \mathsf{match}\ x\ \{\overline{p_i \to e_i}\} \longrightarrow_r \mathsf{H} \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ x;\ e_i[xs{:=}ys]$ with $p_i = C\ xs$ and $(x \mapsto^n C\ ys) \in \mathsf{H}$

$\mathsf{H} \vdash \mathsf{match}\ x\ \{\overline{C\ xs \to e_i}\} \dashv \mathsf{H}'$	given
$\mathsf{H} \vdash x \dashv \mathsf{H}_1$	given
$\mathsf{H}_1, [\![xs]\!] \vdash e_i \dashv \mathsf{H}'$	given
$xs \notin \mathsf{H}_i$	given
$\mathsf{H} = \mathsf{H}_1 + x$	Lemma 3
$(x \mapsto^n C\ ys) \in \mathsf{H}$	given
$ys \in \mathsf{H}$	H ok
$\mathsf{H} \vdash \mathsf{dup}\ ys;\ () \dashv \mathsf{H} + ys$	by definition
$\mathsf{H} + ys = \mathsf{H}_1 + x + ys$	
$= \mathsf{H}_1 + ys + x$	Lemma 5
$\mathsf{H}_1 + ys + x \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}_1 + ys$	Lemma 4
$\mathsf{H}_1 + ys \vdash e_i[xs{:=}ys] \dashv \mathsf{H}'$	by substitution

case (lam_r) $\mathsf{H} \mid (\lambda^{ys}\ x.\ e) \longrightarrow_h \mathsf{H}, f \mapsto^1 \lambda^{ys}\ x.\ e \mid f$ $\quad$ fresh f

$\mathsf{H} \vdash \lambda^{ys}\, x.\, e \dashv \mathsf{H}_1$ given
$\mathsf{H} \vdash ys \dashv \mathsf{H}_1$ given
$\mathsf{H},\, f \mapsto^1 \lambda^{ys}\, x.\, e \vdash f \dashv \mathsf{H}_1$ DRVAR

case (con_r) $\mathsf{H} \mid C\, x_1 \ldots x_n \longrightarrow_r \mathsf{H}, z \mapsto^1 C\, x_1 \ldots x_n \mid z$ fresh z

$\mathsf{H} \vdash C\, x_1 \ldots x_n \dashv \mathsf{H}_1$ given
$\mathsf{H},\, z \mapsto^1 C\, x_1 \ldots x_n \vdash z \dashv \mathsf{H}$ DRCON

case (dup_r) $\mathsf{H}, x \mapsto^n v \mid \mathsf{dup}\, x;\, e \longrightarrow_r \mathsf{H}, x \mapsto^{n+1} v \mid e$

$\mathsf{H},\, x \mapsto^n v \vdash \mathsf{dup}\, x;\, e \dashv \mathsf{H}_1$ given
$\mathsf{H},\, x \mapsto^{n+1} v \vdash e \dashv \mathsf{H}_1$ DRDUP

case $(drop_r)$ $\mathsf{H}, x \mapsto^{n+1} v \mid \mathsf{drop}\, x;\, e \longrightarrow_r \mathsf{H},\, x \mapsto^n v \mid e$ if $n \geq 1$

$\mathsf{H},\, x \mapsto^{n+1} v \vdash \mathsf{drop}\, x;\, e \dashv \mathsf{H}_1$ given
$\mathsf{H},\, x \mapsto^n v \vdash e \dashv \mathsf{H}_1$ $[drdrop]

case $(dlam_r)$ $\mathsf{H}, x \mapsto^1 \lambda^{ys} z.e \mid \mathsf{drop}\, x;\, e \longrightarrow_r \mathsf{H} \mid \mathsf{drop}\, ys;\, e$

$\mathsf{H},\, x \mapsto^1 \lambda^{ys} z.e \vdash \mathsf{drop}\, x;\, e \dashv \mathsf{H}_1$ given
$\mathsf{H} \vdash \mathsf{drop}\, ys;\, e \dashv \mathsf{H}_1$ DRDROPLAM

case $(dcon_r)$ $\mathsf{H}, x \mapsto^1 C\, ys \mid \mathsf{drop}\, x;\, e \longrightarrow_r \mathsf{H} \mid \mathsf{drop}\, ys;\, e$

$\mathsf{H},\, x \mapsto^1 C\, ys \vdash \mathsf{drop}\, x;\, e \dashv \mathsf{H}_1$ given
$\mathsf{H} \vdash \mathsf{drop}\, ys;\, e \dashv \mathsf{H}_1$ DRDROPCON

□

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 $\mathsf{H}_1 \vdash \mathsf{E}[e_1] \dashv \mathsf{H}'$, and $\mathsf{H}_1 \mid \mathsf{E}[e_1] \longmapsto_r \mathsf{H}_2 \mid \mathsf{E}[e_2]$, then $\mathsf{H}_2 \vdash \mathsf{E}[e_2] \dashv \mathsf{H}'$.

Proof. (*Proof for Lemma 15*) By induction on E.
case $\mathsf{E} = \square$. Follows by Lemma 14.
case $\mathsf{E} = \mathsf{E}_1\, e$.

$\mathsf{H}_1 \vdash \mathsf{E}_1[e_1]\, e \dashv \mathsf{H}_2$ given
$\mathsf{H}_1 \vdash \mathsf{E}_1[e_1] \dashv \mathsf{H}_3$ DRAPP
$\mathsf{H}_3 \vdash e \dashv \mathsf{H}_2$ given
$\mathsf{H}_1 \vdash \mathsf{E}_1[e_2] \dashv \mathsf{H}_3$ I.H.
$\mathsf{H}_1 \vdash \mathsf{E}_1[e_2]\, e \dashv \mathsf{H}_2$ DRAPP

case $\mathsf{E} = x\, \mathsf{E}_1$.

$\mathsf{H}_1 \vdash x\, \mathsf{E}_1[e_1] \dashv \mathsf{H}_2$ given
$\mathsf{H}_1 \vdash x \dashv \mathsf{H}_3$ DRAPP
$\mathsf{H}_3 \vdash \mathsf{E}_1[e_1] \dashv \mathsf{H}_2$ given
$\mathsf{H}_3 \vdash \mathsf{E}_1[e_2] \dashv \mathsf{H}_2$ I.H.
$\mathsf{H}_1 \vdash x\, \mathsf{E}_1[e_2] \dashv \mathsf{H}_2$ DRAPP

case $\mathsf{E} = \mathsf{val}\, x = \mathsf{E}_1;\, e$.

$\mathsf{H}_1 \vdash \mathsf{val}\, x = \mathsf{E}_1[e_1];\, e \dashv \mathsf{H}_2$ given
$\mathsf{H}_1 \vdash \mathsf{E}_1[e_1] \dashv \mathsf{H}_3$ DRBIND
$\mathsf{H}_3,\, x \mapsto^1 () \vdash e \dashv \mathsf{H}_2$ given
$x \notin \mathsf{H}_2$ given
$\mathsf{H}_1 \vdash \mathsf{E}_1[e_2] \dashv \mathsf{H}_3$ I.H.
$\mathsf{H}_1 \vdash \mathsf{val}\, x = \mathsf{E}_1[e_2]\,;\, e \dashv \mathsf{H}_2$ DRBIND

case $\mathsf{E} = C\, x_1 \ldots x_i\, \mathsf{E}_1\, v_j \ldots v_n$.

$$
\begin{array}{ll}
\mathsf{H}_1 \;\vdash\; C\ x_1 \ldots x_i\ \mathsf{E}_1[e_1]\ v_j \ldots v_n \;\dashv\; \mathsf{H}_2 & \text{given} \\
\mathsf{H}_1 \;\vdash\; x_1 \ldots x_i \;\dashv\; \mathsf{H}_3 & \textsc{drcon} \\
\mathsf{H}_3 \;\vdash\; \mathsf{E}_1[e_1] \;\dashv\; \mathsf{H}_4 & \text{given} \\
\mathsf{H}_4 \;\vdash\; v_j \ldots v_n \;\dashv\; \mathsf{H}_2 & \text{given} \\
\mathsf{H}_3 \;\vdash\; \mathsf{E}_1[e_2] \;\dashv\; \mathsf{H}_4 & \text{I.H.} \\
\mathsf{H}_1 \;\vdash\; C\ x_1 \ldots x_i\ \mathsf{E}_1[e_2]\ v_j \ldots v_n \;\dashv\; \mathsf{H}_2 & \textsc{drcon}
\end{array}
$$

□

Theorem 6. (*No garbage*)
Given $\varnothing;\varnothing \vdash\ e\ \rightsquigarrow e_1$, and $\varnothing \mid e_1 \longmapsto^*{}_r \mathsf{H}_i \mid e_i$, then $\mathsf{H}_i\ \vdash\ e_i\ \dashv\ \varnothing$.

Proof. (*Of Theorem 6*)

$$
\begin{array}{ll}
\varnothing\ \mathsf{ok} & \text{by construction} \\
e_1\ \mathsf{ok} & \text{by } \textsc{lam} \\
\varnothing \;\vdash\; e_1 \;\dashv\; \varnothing & \text{Lemma 7} \\
\mathsf{H}_i \;\vdash\; e_i \;\dashv\; \varnothing & \text{Lemma 15}
\end{array}
$$

□

D.3.3 No Garbage.

Lemma 16. (*Reachability*)
If $\mathsf{H}_1\ \vdash\ e\ \dashv\ \mathsf{H}_2$, then there for all $y\ \in \mathsf{dom}(\mathsf{H}_1)\ -\ \mathsf{dom}(\mathsf{H}_2)$, $\mathsf{reach}(y,\ \mathsf{H}_1 \mid e)$. For ease of reference, we denote it as $\mathsf{reach}(\mathsf{H}_1\ -\ \mathsf{H}_2,\ \mathsf{H}_1 \mid e)$

Proof. (*Of Lemma 16*) By induction on the judgment.

case

$$
\begin{array}{ll}
\mathsf{H}_0 \;\vdash\; C\ v_1 \ldots v_n \;\dashv\; \mathsf{H}_n & \text{given} \\
\mathsf{H}_0 \;\vdash\; v_1 \;\dashv\; \mathsf{H}_1 \ldots \mathsf{H}_{n-1} \;\vdash\; v_n \;\dashv\; \mathsf{H}_n & \textsc{drcon} \\
\mathsf{reach}(\mathsf{H}_{i-1}\ -\ \mathsf{H}_i,\ \mathsf{H}_{i-1} \mid v_i) & \text{I.H.} \\
\mathsf{reach}(\mathsf{H}_{i-1}\ -\ \mathsf{H}_i,\ \mathsf{H}_0 \mid v_i) & \text{Lemma 2} \\
\mathsf{reach}(\mathsf{H}_n\ -\ \mathsf{H}_0,\ \mathsf{H}_0 \mid C\ v_1 \ldots v_n) & \text{Follows}
\end{array}
$$

case

$$
\begin{array}{ll}
\mathsf{H},\ x \mapsto^{n+1}\ v \;\vdash\; x \;\dashv\; \mathsf{H},\ x \mapsto^{n}\ v & \text{given} \\
\mathsf{dom}(\mathsf{H},\ x \mapsto^{n+1}\ v)\ -\ \mathsf{dom}(\mathsf{H},\ x \mapsto^{n}\ v)\ =\ \varnothing &
\end{array}
$$

case

$$
\begin{array}{ll}
\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e \;\vdash\; x \;\dashv\; \mathsf{H}_1 & \text{given} \\
\mathsf{H} \;\vdash\; ys \;\dashv\; \mathsf{H}_1 & \textsc{drvarlam} \\
\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid ys) & \text{I.H.} \\
\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lambda^{ys} z.\ e) & \text{by definition} \\
\mathsf{reach}((\mathsf{H}, x \mapsto^{1} \lambda^{ys} z.\ e) - \mathsf{H}_1,\ (\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e) \mid x) & \text{follows}
\end{array}
$$

case

$$
\begin{array}{ll}
\mathsf{H},\ x \mapsto^{1}\ C\ ys \;\vdash\; x \;\dashv\; \mathsf{H}_1 & \text{given} \\
\mathsf{H} \;\vdash\; ys \;\dashv\; \mathsf{H}_1 & \textsc{drvarcon} \\
\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid ys) & \text{I.H.} \\
\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid C\ ys) & \text{by definition} \\
\mathsf{reach}((\mathsf{H}, x \mapsto^{1}\ C\ ys) - \mathsf{H}_1,\ (\mathsf{H}, x \mapsto^{1}\ C\ ys)\ \mid x) & \text{follows}
\end{array}
$$

case

$\mathsf{H},\ x \mapsto^{n} v \ \vdash\ \mathsf{dup}\ x;\ e\ \dashv\ \mathsf{H}_1$	given
$\mathsf{H},\ x \mapsto^{n+1} v \ \vdash\ e\ \dashv\ \mathsf{H}_1$	DRDUP
$\mathsf{reach}((\mathsf{H}, x \mapsto^{n+1} v) - \mathsf{H}_1,\ (\mathsf{H}, x \mapsto^{n+1} v) \mid e)$	I.H.
$\mathsf{reach}((\mathsf{H}, x \mapsto^{n} v) - \mathsf{H}_1,\ (\mathsf{H}, x \mapsto^{n} v) \mid \mathsf{dup}\ x;\ e)$	follows

case

$\mathsf{H},\ x \mapsto^{n+1} v \ \vdash\ \mathsf{drop}\ x;\ e\ \dashv\ \mathsf{H}_1$	given
$\mathsf{H},\ x \mapsto^{n} v \ \vdash\ e\ \dashv\ \mathsf{H}_1$	DRDROP
$\mathsf{reach}((\mathsf{H},\ x \mapsto^{n+1} v) - \mathsf{H}_1,\ (\mathsf{H}, x \mapsto^{n+1} v) \mid e)$	I.H.
$\mathsf{reach}((\mathsf{H},\ x \mapsto^{n} v) - \mathsf{H}_1,\ (\mathsf{H}, x \mapsto^{n} v) \mid \mathsf{drop}\ x;\ e)$	follows

case

$\mathsf{H},\ x \mapsto^{1} C\ ys \ \vdash\ \mathsf{drop}\ x;\ e\ \dashv\ \mathsf{H}_1$	given
$\mathsf{H}\ \vdash\ \mathsf{drop}\ ys;\ e\ \dashv\ \mathsf{H}_1$	DRDROPCON
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \mathsf{drop}\ ys;\ e)$	I.H.
$\mathsf{reach}((\mathsf{H},\ x \mapsto^{1} C\ ys) - \mathsf{H}_1,\ (\mathsf{H},\ x \mapsto^{1} C\ ys) \mid \mathsf{drop}\ x;\ e)$	follows

case

$\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e \ \vdash\ \mathsf{drop}\ x;\ e\ \dashv\ \mathsf{H}_1$	given
$\mathsf{H}\ \vdash\ \mathsf{drop}\ ys;\ e\ \dashv\ \mathsf{H}_1$	DRDROPLAM
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \mathsf{drop}\ ys;\ e)$	I.H.
$\mathsf{reach}((\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e) - \mathsf{H}_1,\ (\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e) \mid \mathsf{drop}\ x;\ e)$	follows

case

$\mathsf{H}\ \vdash\ \lambda^{ys}\ x.\ e\ \dashv\ \mathsf{H}_1$	given
$\mathsf{H} \vdash\ ys\ \dashv \mathsf{H}_1$	DRLAM
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid ys)$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lambda^{ys} x.\ e)$	follows

case

$\mathsf{H}\ \vdash\ e_1\ e_2\ \dashv\ \mathsf{H}_2$	given
$\mathsf{H}\ \vdash\ e_1\ \dashv\ \mathsf{H}_1$	DRAPP
$\mathsf{H}_1\ \vdash\ e_2\ \dashv\ \mathsf{H}_2$	DRAPP
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid e_1)$	I.H.
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H}_1 \mid e_2)$	I.H.
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H} \mid e_2)$	Lemma 2
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_2,\ \mathsf{H} \mid e_1\ e_2)$	follows

case

$\mathsf{H}\ \vdash\ \mathsf{val}\ x\ =\ e_1\ ;\ e_2\ \dashv\ \mathsf{H}_2$	given
$\mathsf{H}\ \vdash\ e_1\ \dashv\ \mathsf{H}_1$	DRBIND
$\mathsf{H}_1, x \mapsto^{1} () \vdash\ e_2\ \dashv\ \mathsf{H}_2$	DRBIND
$x \notin \mathsf{H},\ \mathsf{H}_2$	DRBIND
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid e_1)$	I.H.
$\mathsf{reach}((\mathsf{H}_1, x \mapsto^{1} ()) - \mathsf{H}_2,\ (\mathsf{H}_1, x \mapsto^{1} ()) \mid e_2)$	I.H.
$\mathsf{reach}((\mathsf{H}_1, x \mapsto^{1} ()) - \mathsf{H}_2,\ (\mathsf{H}, x \mapsto^{1} ()) \mid e_2)$	Lemma 2
$\mathsf{dom}(\mathsf{H}_1) \subseteq \mathsf{dom}(\mathsf{H}_1, x \mapsto^{1} ())$	
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ (\mathsf{H}, x \mapsto^{1} ()) \mid e_2)$	follows
$x \notin \mathsf{H}$	known
$x \notin \mathsf{dom}(\mathsf{H}) - \mathsf{dom}(\mathsf{H}_2)$	follows
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_2,\ \mathsf{H} \mid \mathsf{val}\ x\ =\ e_1;\ e_2)$	follows

case

$\mathsf{H} \ \vdash \ \mathsf{match}\ x\ \{\ \overline{p_i \ \mapsto e_i}\ \} \ \dashv \ \mathsf{H}'$	given
$\mathsf{H} \ \vdash \ x \ \dashv \ \mathsf{H}_1$	DRMATCH
$\mathsf{H}_1,\ [\![\mathsf{bv}(p_i)]\!] \ \vdash \ e_i \ \dashv \ \mathsf{H}'$	DRMATCH
$\mathsf{bv}(p_i) \ \notin \mathsf{H}, \mathsf{H}'$	DRMATCH
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid x)$	I.H.
$\mathsf{reach}((\mathsf{H}_1, [\![\mathsf{bv}(p_i)]\!]) - \ \mathsf{H}',\ (\mathsf{H}_1, [\![\mathsf{bv}(p_i)]\!]) \ \mid e_i)$	I.H.
$\mathsf{reach}((\mathsf{H}_1, [\![\mathsf{bv}(p_i)]\!]) - \ \mathsf{H}',\ (\mathsf{H}, [\![\mathsf{bv}(p_i)]\!]) \ \mid e_i)$	Lemma 2
$\mathsf{dom}(\mathsf{H}_1) \ \subseteq \mathsf{dom}(\mathsf{H}_1, [\![\mathsf{bv}(p_i)]\!])$	
$\mathsf{reach}(\mathsf{H}_1 - \ \mathsf{H}',\ (\mathsf{H}, [\![\mathsf{bv}(p_i)]\!]) \ \mid e_i)$	follows
$\mathsf{bv}(p_i) \ \notin \mathsf{H}$	known
$\mathsf{bv}(p_i) \ \notin \mathsf{dom}(\mathsf{H}) - \mathsf{dom}(\mathsf{H}')$	follows
$\mathsf{reach}(\mathsf{H} - \mathsf{H}',\ \mathsf{H} \mid \mathsf{match}\ x\ \{\ \overline{p_i \ \mapsto e_i}\ \}\ )$	follows

□

Proof. (*Of Theorem 2*)

$\varnothing ; \varnothing \ \vdash \ e \ \rightsquigarrow e'$	given
$Hi \ \vdash \ e_i \ \dashv \varnothing$	Theorem 6
$\mathsf{reach}(\mathsf{H}_i \ - \ \varnothing,\ \mathsf{H}_i \mid e_i)$	Lemma 16

□

D.4 Soundness of Syntax-directed Translation

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

$\Delta \mid x \vdash_{\mathsf{s}} \ x \ \rightsquigarrow x$	given
$\Delta \mid x \ \vdash \ x \ \rightsquigarrow x$	VAR

case

$\Delta, x \mid \varnothing \vdash_{\mathsf{s}} \ x \ \rightsquigarrow \mathsf{dup}\ x;\ x$	given
$\Delta, x \mid x \ \vdash \ x \ \rightsquigarrow x$	VAR
$\Delta, x \mid \varnothing \ \vdash \ x \ \rightsquigarrow \mathsf{dup}\ x;\ x$	DUP

case

$\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \ \lambda x.\ e \ \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ e'$	given
$x \ \in \mathsf{fv}(e)$	SLAM
$\varnothing \mid ys, x \vdash_{\mathsf{s}} \ e \ \rightsquigarrow e'$	SLAM
$ys \ = \ \mathsf{fv}(\lambda x.\ e)$	SLAM
$\Delta_1 \ = \ ys - \Gamma$	SLAM
$\varnothing \mid ys, x \ \vdash \ e \ \rightsquigarrow e'$	I.H.
$\Delta, \Delta_1 \mid \Gamma,\ \Delta_1 \vdash_{\mathsf{s}} \ \lambda x.\ e \ \rightsquigarrow \lambda^{ys}\ x.\ e'$	LAM
$\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \ \lambda x.\ e \ \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ e'$	DUP

case

$\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \ \lambda x.\ e \ \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ (\mathsf{drop}\ x;\ e')$	given
$x \notin \mathsf{fv}(e)$	SLAM-DROP
$\varnothing \mid ys \vdash_{\mathsf{s}} \ e \ \rightsquigarrow e'$	SLAM-DROP
$ys \ = \ \mathsf{fv}(\lambda x.\ e)$	SLAM-DROP
$\Delta_1 \ = \ ys - \Gamma$	SLAM-DROP
$\varnothing \mid ys \ \vdash \ e \ \rightsquigarrow e'$	I.H.
$\varnothing \mid ys,\ x \ \vdash \ e \ \rightsquigarrow \mathsf{drop}\ x;\ e'$	DROP
$ys \ = \ \mathsf{fv}(\lambda x.\ \mathsf{drop}\ x;\ e)$	follows
$\Delta, \Delta_1 \mid \Gamma,\ \Delta_1 \vdash_{\mathsf{s}} \ \lambda x.\ e \ \rightsquigarrow \lambda^{ys}\ x.\ \mathsf{drop}\ x;\ e'$	LAM
$\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \ \lambda x.\ e \ \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ e'$	DUP

case

$\Delta \mid \Gamma \vdash_s e_1\ e_2 \rightsquigarrow e_1'\ e_2'$	given
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_s e_1 \rightsquigarrow e_1'$	SAPP
$\Delta \mid \Gamma_2 \vdash_s e_2 \rightsquigarrow e_2'$	SAPP
$\Gamma_2 \mathrel{\hat{=}} \Gamma \cap \mathsf{fv}(e_2)$	SAPP
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash e_1 \rightsquigarrow e_1'$	I.H.
$\Delta \mid \Gamma_2 \vdash e_2 \rightsquigarrow e_2'$	I.H.
$\Delta \mid \Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2'$	APP

case

$\Delta \mid \Gamma \vdash_s \mathsf{val}\ x = e_1;\ e_2 \rightsquigarrow \mathsf{val}\ x = e_1';\ e_2'$	given
$x \in \mathsf{fv}(e_2)$	SBIND
$x \notin \Delta, \Gamma$	SBIND
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_s e_1 \rightsquigarrow e_1'$	SBIND
$\Delta \mid \Gamma_2, x \vdash_s e_2 \rightsquigarrow e_2'$	SBIND
$\Gamma_2 \mathrel{\hat{=}} \Gamma \cap (\mathsf{fv}(e_2) - x)$	SBIND
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash e_1 \rightsquigarrow e_1'$	I.H.
$\Delta \mid \Gamma_2, x \vdash e_2 \rightsquigarrow e_2'$	I.H.
$\Delta \mid \Gamma \vdash \mathsf{val}\ x = e_1;\ e_2 \rightsquigarrow \mathsf{val}\ x = e_1';\ e_2'$	BIND

case

$\Delta \mid \Gamma \vdash_s \mathsf{val}\ x = e_1;\ e_2 \rightsquigarrow \mathsf{val}\ x = e_1';\ \mathsf{drop}\ x;\ e_2'$	given
$x \notin \mathsf{fv}(e_2),\ \Delta,\ \Gamma$	SBIND-DROP
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_s e_1 \rightsquigarrow e_1'$	SBIND-DROP
$\Delta \mid \Gamma_2 \vdash_s e_2 \rightsquigarrow e_2'$	SBIND-DROP
$\Gamma_2 \mathrel{\hat{=}} \Gamma \cap \mathsf{fv}(e_2)$	SBIND-DROP
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash e_1 \rightsquigarrow e_1'$	I.H.
$\Delta \mid \Gamma_2 \vdash e_2 \rightsquigarrow e_2'$	I.H.
$\Delta \mid \Gamma_2,\ x \vdash e_2 \rightsquigarrow \mathsf{drop}\ x;\ e_2'$	DROP
$\Delta \mid \Gamma \vdash \mathsf{val}\ x = e_1;\ e_2 \rightsquigarrow \mathsf{val}\ x = e_1';\ \mathsf{drop}\ x;\ e_2'$	BIND

case

$\Delta \mid \Gamma, x \vdash_s \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \rightsquigarrow \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{drop}\ \Gamma_i';\ e_i'}\ \}$	given
$\Delta \mid \Gamma_i \vdash_s e_i \rightsquigarrow e_i'$	SMATCH
$\Gamma_i \mathrel{\hat{=}} (\Gamma, \mathsf{bv}(p_i)) \cap \mathsf{fv}(e_i)$	SMATCH
$\Gamma_i' = (\Gamma, \mathsf{bv}(p_i)) - \Gamma_i$	SMATCH
$\Delta \mid \Gamma_i \vdash e_i \rightsquigarrow e_i'$	I.H.
$\Delta \mid \Gamma_i,\ \Gamma_i' \vdash e_i \rightsquigarrow \mathsf{drop}\ \Gamma_i';\ e_i'$	DROP
$\Delta \mid \Gamma,\ \mathsf{bv}(p_i) \vdash e_i \rightsquigarrow \mathsf{drop}\ \Gamma_i';\ e_i'$	by substitution
$\Delta \mid \Gamma, x \vdash \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \rightsquigarrow \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{drop}\ \Gamma_i';\ e_i'}\ \}$	MATCH

case

$\Delta \mid \Gamma \vdash_s C\ v_1 \ldots v_n \rightsquigarrow C\ v_1' \ldots v_n'$	given
$\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \vdash_s v_i \rightsquigarrow v_i'$	SCON
$\Gamma_i \mathrel{\hat{=}} (\Gamma - \Gamma_{i+1} - \ldots - \Gamma_n) \cap \mathsf{fv}(v_i)$	SCON
$1 \leqslant i \leqslant n$	SCON
$\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \vdash v_i \rightsquigarrow v_i'$	I.H.
$\Delta \mid \Gamma \vdash C\ v_1 \ldots v_n \rightsquigarrow C\ v_1' \ldots v_n'$	CON

□

D.5 Precision

D.5.1 A Heap Reference Counting Calculus for the Algorithm.

$$\frac{\mathsf{H} \vdash_s v_1 \dashv \mathsf{H}_1 \ldots \mathsf{H}_{n-1} \vdash_s v_n \dashv \mathsf{H}_n}{\mathsf{H} \vdash_s C\ v_1 \ldots v_n \dashv \mathsf{H}_1}\ [\textsc{srcon}]$$

$$\frac{}{\mathsf{H},\ x \mapsto^{n+1} v \vdash_s x \dashv \mathsf{H},\ x \mapsto^{n} v}\ [\textsc{srvar}]$$

$$\frac{\mathsf{H} \vdash_s ys \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e \vdash_s x \dashv \mathsf{H}_1}\ [\textsc{srvarlam}]$$

$$\frac{\mathsf{H} \vdash_s ys \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} C\ ys \vdash_s x \dashv \mathsf{H}_1}\ [\textsc{srvarcon}]$$

$$\frac{\mathsf{H} \vdash_s ys \dashv \mathsf{H}_1 \quad ys \mapsto^1 (),\ x \mapsto^1 () \vdash_s e \dashv \varnothing}{\mathsf{H} \vdash_s \lambda^{ys}\ x.\ e \dashv \mathsf{H}_1}\ [\textsc{srlam1}]$$

$$\frac{\mathsf{H} \vdash_s ys \dashv \mathsf{H}_1 \quad ys \mapsto^1 () \vdash_s e \dashv \varnothing}{\mathsf{H} \vdash_s \lambda^{ys}\ x.\ \mathsf{drop}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{srlam2}]$$

$$\frac{\mathsf{H} \vdash_s e_1 \dashv \mathsf{H}_1 \quad \mathsf{H}_1 \vdash_s e_2 \dashv \mathsf{H}_2}{\mathsf{H} \vdash_s e_1\ e_2 \dashv \mathsf{H}_2}\ [\textsc{srapp}]$$

$$\frac{\mathsf{H} \vdash_s e_1 \dashv \mathsf{H}_1 \quad \mathsf{H}_1, x \mapsto^1 () \vdash_s e_2 \dashv \mathsf{H}_2 \quad x \notin \mathsf{H}, \mathsf{H}_2}{\mathsf{H} \vdash_s \mathsf{val}\ x\ =\ e_1\ ;\ e_2 \dashv \mathsf{H}_2}\ [\textsc{srbind1}]$$

$$\frac{\mathsf{H} \vdash_s e_1 \dashv \mathsf{H}_1 \quad \mathsf{H}_1 \vdash_s e_2 \dashv \mathsf{H}_2 \quad x \notin \mathsf{H}}{\mathsf{H} \vdash_s \mathsf{val}\ x\ =\ e_1\ ;\ \mathsf{drop}\ x;\ e_2 \dashv \mathsf{H}_2}\ [\textsc{srbind2}]$$

$$\frac{\begin{array}{l}\mathsf{H} \vdash_s x \dashv \mathsf{H}_1 \quad \mathsf{H}_1,\ [\![\mathsf{bv}(p_i)]\!] \vdash_r \mathsf{drop}\ ys_i;\ () \dashv \mathsf{H}_i \\ \mathsf{H}_i \vdash_s e_i \dashv \mathsf{H}' \quad \mathsf{bv}(p_i) \notin \mathsf{H}, \mathsf{H}' \\ ys_i \subseteq \mathsf{fv}(\overline{\lceil e_i \rceil}) \cup \mathsf{bv}(p_i)\end{array}}{\mathsf{H} \vdash_s \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{drop}\ ys_i;\ e_i}\ \} \dashv \mathsf{H}'}\ [\textsc{srmatch}]$$

$$\frac{\mathsf{H},\ x \mapsto^{n+1} v \vdash_s e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{n} v \vdash_s \mathsf{dup}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{srdup}]$$

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

$$\frac{\mathsf{H},\ x \mapsto^{n+1} v \vdash_r e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{n} v \vdash_r \mathsf{dup}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{rrdup}]$$

$$\frac{\mathsf{H},\ x \mapsto^{n} v \vdash_r e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{n+1} v \vdash_r \mathsf{drop}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{rrdrop}]$$

$$\frac{\mathsf{H} \vdash_r e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} () \vdash_r \mathsf{drop}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{rrdropunit}]$$

$$\frac{\mathsf{H} \vdash_r \mathsf{drop}\ ys;\ e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} \lambda^{ys} z.\ e \vdash_r \mathsf{drop}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{rrdroplam}]$$

$$\frac{\mathsf{H} \vdash_r \mathsf{drop}\ ys;\ e \dashv \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1} C\ ys \vdash_r \mathsf{drop}\ x;\ e \dashv \mathsf{H}_1}\ [\textsc{rrdropcons}]$$

$$\frac{\mathsf{H} \ \vdash_s e \ \dashv \mathsf{H}_1}{\mathsf{H} \ \vdash_r e \ \dashv \mathsf{H}_1}\ [\textsc{rrsub}]$$

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 ok_r, then all $(\lambda^{ys}\ x.e_1)$ in e satisfies (1) $[\![ys,\ x]\!] \ \vdash_r e_1 \ \dashv \varnothing$; or (2) $e_1 \ = \ \mathsf{drop}\ x;\ e_2$, and $[\![ys]\!] \ \vdash_r e_2 \ \dashv \varnothing$.

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

Lemma 17. (*Heap Reference Counting Free variables*)
If $\mathsf{H}_1 \ \vdash_r e \ \dashv \mathsf{H}_2$ or $\mathsf{H}_1 \ \vdash_s e \ \dashv \mathsf{H}_2$, then $\mathsf{fv}(e) \ \in \mathsf{H}_1$, and $\mathsf{fv}(\mathsf{H}_2) \ \in \mathsf{H}_1$ with same domains.

Lemma 18. (*Drop is dual to extension*)
If $\mathsf{H}_1 \ \vdash_r \mathsf{drop}\ x;\ () \ \dashv \mathsf{H}_2$, then $\mathsf{H}_1 \ = \ \mathsf{H}_2 +\!\!\!+\ x$. Similarly, if $\mathsf{H}_1 \ \vdash_r x \ \dashv \mathsf{H}_2$ or $\mathsf{H}_1 \ \vdash_s x \ \dashv \mathsf{H}_2$, then $\mathsf{H}_1 \ = \ \mathsf{H}_2 +\!\!\!+\ x$.

Lemma 19. (*Extension is dual to drop*)
$\mathsf{H} +\!\!\!+\ x \ \vdash_r \mathsf{drop}\ x;\ () \ \dashv \mathsf{H}$. Similarly, $\mathsf{H} +\!\!\!+\ x \ \vdash_s x \ \dashv \mathsf{H}$ and $\mathsf{H} +\!\!\!+\ x \ \vdash_s x \ \dashv \mathsf{H}$.

Lemma 20.
$[\![\Gamma,\ x]\!] \ \vdash_r x \ \dashv [\![\Gamma]\!]$. Similarly, if $[\![\Delta]\!] \ \vdash_r e \ \dashv \mathsf{H}$, then $[\![\Gamma, x]\!] \ \vdash_r \mathsf{drop}\ x;\ e \ \dashv \mathsf{H}$.

Theorem 7. (*No garbage*)
Given $\varnothing; \varnothing \vdash_r e \ \rightsquigarrow e_1$, and $\varnothing \mid e_1 \longmapsto^*{}_r \mathsf{H}_i \mid e_i$, then $\mathsf{H}_i \ \vdash_r e_i \ \dashv \varnothing$.

D.5.3 Relating to linear resource calculus.

Lemma 21. (*Algorithmic linear resource calculus relates to reference counting*)
If $\Delta \mid \Gamma \ \vdash_s e \ \rightsquigarrow e'$, then $[\![\Delta, \Gamma]\!] \ \vdash_s e' \ \dashv [\![\Delta]\!]$. By $\textsc{rrsub}$ we also have $[\![\Delta, \Gamma]\!] \ \vdash_r e' \ \dashv [\![\Delta]\!]$.

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

$\Delta \mid x \ \vdash_s x \ \rightsquigarrow x$	given
$[\![\Delta,\ x]\!] \ \vdash_s x \ \dashv [\![\Delta]\!]$	Lemma 20

case

$\Delta, x \mid \varnothing \ \vdash_s x \ \rightsquigarrow \mathsf{dup}\ x;\ x$	given
$[\![\Delta,\ x]\!] +\!\!\!+\ x \ \vdash_s x \ \dashv [\![\Delta, x]\!]$	Lemma 19
$[\![\Delta,\ x]\!] \ \vdash_s \mathsf{dup}\ x;\ x \ \dashv [\![\Delta, x]\!]$	$\textsc{srdup}$

case

$\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \lambda x.\ e \ \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ e'$	given
$x \ \in \mathsf{fv}(e)$	$\textsc{slam}$
$\varnothing \mid ys, x \vdash_{\mathsf{s}} e \ \rightsquigarrow e'$	$\textsc{slam}$
$ys \ \hat{=} \ \mathsf{fv}(\lambda x.\ e)$	$\textsc{slam}$
$\Delta_1 \ = \ ys - \Gamma$	$\textsc{slam}$
$[\![ys, x]\!] \ \vdash_s e' \ \dashv \varnothing$	I.H.
$[\![\Delta,\ \Delta_1,\ ys]\!] \ \vdash_s ys \ \dashv [\![\Delta, \Delta_1]\!]$	Lemma 19
$[\![\Delta,\ \Delta_1,\ \Gamma,\ \Delta_1]\!] \ \vdash_s ys \ \dashv [\![\Delta, \Delta_1]\!]$	by substitution
$[\![\Delta,\ \Delta_1,\ \Gamma,\ \Delta_1]\!] \ \vdash_s \lambda^{ys}\ x.\ e' \ \dashv [\![\Delta, \Delta_1]\!]$	$\textsc{srlam1}$
$[\![\Delta,\ \Delta_1,\ \Gamma]\!] \ \vdash_s \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ e' \ \dashv [\![\Delta, \Delta_1]\!]$	$\textsc{srdup}$

case

$\Delta, \Delta_1 \mid \Gamma \vdash_{\mathsf{s}} \lambda x.\ e \ \rightsquigarrow \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ (\mathsf{drop}\ x;\ e')$	given
$x \notin \mathsf{fv}(e)$	SLAM-DROP
$\varnothing \mid ys \vdash_{\mathsf{s}} e \ \rightsquigarrow e'$	SLAM-DROP
$ys \ \hat{=}\ \mathsf{fv}(\lambda x.\ e)$	SLAM-DROP
$\Delta_1 \ = \ ys - \Gamma$	SLAM-DROP
$[\![ys]\!] \ \vdash_s e' \ \dashv \varnothing$	I.H.
$[\![\Delta,\ \Delta_1,\ ys]\!] \ \vdash_s ys \ \dashv [\![\Delta, \Delta_1]\!]$	Lemma 19
$[\![\Delta,\ \Delta_1,\ \Gamma,\ \Delta_1]\!] \ \vdash_s ys \ \dashv [\![\Delta, \Delta_1]\!]$	by substitution
$[\![\Delta,\ \Delta_1,\ \Gamma,\ \Delta_1]\!] \ \vdash_s \lambda^{ys}\ x.\ \mathsf{drop}\ x;\ e \ \dashv [\![\Delta, \Delta_1]\!]$	SRLAM2
$[\![\Delta,\ \Delta_1,\ \Gamma]\!] \ \vdash_s \mathsf{dup}\ \Delta_1;\ \lambda^{ys}\ x.\ \mathsf{drop}\ x;\ e' \ \dashv [\![\Delta, \Delta_1]\!]$	SRDUP

case

$\Delta \mid \Gamma \vdash_{\mathsf{s}} e_1\ e_2 \ \rightsquigarrow e_1'\ e_2'$	given
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \ \rightsquigarrow e_1'$	SAPP
$\Delta \mid \Gamma_2 \vdash_{\mathsf{s}} e_2 \ \rightsquigarrow e_2'$	SAPP
$\Gamma_2 \ \hat{=}\ \Gamma \cap \mathsf{fv}(e_2)$	SAPP
$[\![\Delta, \Gamma]\!] \ \vdash_s e_1' \ \dashv [\![\Delta,\ \Gamma_2]\!]$	I.H.
$[\![\Delta,\ \Gamma_2]\!] \ \vdash_s e_2' \ \dashv [\![\Delta]\!]$	I.H.
$[\![\Delta, \Gamma]\!] \ \vdash_s e_1'\ e_2' \ \dashv [\![\Delta]\!]$	SRAPP

case

$\Delta \mid \Gamma \vdash_{\mathsf{s}} \mathsf{val}\ x \ = \ e_1;\ e_2 \ \rightsquigarrow \mathsf{val}\ x \ = \ e_1';\ e_2'$	given
$x \ \in \mathsf{fv}(e_2)$	SBIND
$x \notin \Delta, \Gamma$	SBIND
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \ \rightsquigarrow e_1'$	SBIND
$\Delta \mid \Gamma_2, x \vdash_{\mathsf{s}} e_2 \ \rightsquigarrow e_2'$	SBIND
$\Gamma_2 \ \hat{=}\ \Gamma \cap (\mathsf{fv}(e_2) \ - \ x)$	SBIND
$[\![\Delta, \Gamma]\!] \ \vdash_s e_1' \ \dashv [\![\Delta, \Gamma_2]\!]$	I.H.
$[\![\Delta,\ \Gamma_2,\ x]\!] \ \vdash_s e_2' \ \dashv [\![\Delta]\!]$	I.H.
$x \notin [\![\Delta]\!]$	follows
$x \notin [\![\Delta, \Gamma]\!]$	follows
$[\![\Delta, \Gamma]\!] \ \vdash_s \mathsf{val}\ x \ = \ e_1';\ e_2' \ \dashv [\![\Delta]\!]$	SRBIND1

case

$\Delta \mid \Gamma \vdash_{\mathsf{s}} \mathsf{val}\ x \ = \ e_1;\ e_2 \ \rightsquigarrow \mathsf{val}\ x \ = \ e_1';\ \mathsf{drop}\ x;\ e_2'$	given
$x \notin \mathsf{fv}(e_2),\ \Delta,\ \Gamma$	SBIND-DROP
$\Delta, \Gamma_2 \mid \Gamma - \Gamma_2 \vdash_{\mathsf{s}} e_1 \ \rightsquigarrow e_1'$	SBIND-DROP
$\Delta \mid \Gamma_2 \vdash_{\mathsf{s}} e_2 \ \rightsquigarrow e_2'$	SBIND-DROP
$\Gamma_2 \ \hat{=}\ \Gamma \cap \mathsf{fv}(e_2)$	SBIND-DROP
$[\![\Delta, \Gamma]\!] \ \vdash_s e_1' \ \dashv [\![\Delta, \Gamma_2]\!]$	I.H.
$[\![\Delta,\ \Gamma_2]\!] \ \vdash_s e_2' \ \dashv [\![\Delta]\!]$	I.H.
$x \notin [\![\Delta, \Gamma]\!]$	follows
$[\![\Delta, \Gamma]\!] \ \vdash_s \mathsf{val}\ x \ = \ e_1';\ \mathsf{drop}\ x;\ e_2' \ \dashv [\![\Delta]\!]$	SRBIND2

case

$\Delta \mid \Gamma, x \vdash_s \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \rightsquigarrow \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{drop}\ \Gamma_i';\ e_i'}\ \}$	given
$\Delta \mid \Gamma_i \vdash_s e_i \rightsquigarrow e_i'$	SMATCH
$\Gamma_i \ \hat{=}\ (\Gamma, \mathsf{bv}(p_i)) \cap \mathsf{fv}(e_i)$	SMATCH
$\Gamma_i' = (\Gamma, \mathsf{bv}(p_i)) - \Gamma_i$	SMATCH
$[\![\Delta, \Gamma, x]\!] \vdash x \dashv [\![\Delta, \Gamma]\!]$	Lemma 20
$[\![\Delta, \Gamma_i', \Gamma_i]\!] \vdash \mathsf{drop}\ \Gamma_i'\ ; () \dashv [\![\Delta, \Gamma_i]\!]$	Lemma 20
$[\![\Delta, \Gamma_i]\!] \vdash e_i' \dashv [\![\Delta]\!]$	I.H.
$\mathsf{bv}(p_i) \notin [\![\Delta, \Gamma, x]\!]$	assume $\mathsf{bv}(p_i)$ fresh
$\mathsf{bv}(p_i) \notin [\![\Delta]\!]$	follows
$\Gamma_i' = (\Gamma, \mathsf{bv}(p_i)) - \Gamma_i$	known
$\Gamma_i' \subseteq (\Gamma, \mathsf{bv}(p_i))$	follows
$\Gamma, x \subseteq \mathsf{fv}(\mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \})$	invariant
$\Gamma, x \subseteq (x,\ \mathsf{fv}(\overline{e_i}))$	definition of fv
multiplicity of each member in Γ, x is 1	invariant
$\Gamma \subseteq \mathsf{fv}(\overline{e_i})$	follows
$\mathsf{fv}(\overline{\lceil e_i' \rceil}) = \mathsf{fv}(\overline{e_i})$	Lemma 13
$\Gamma \subseteq \mathsf{fv}(\overline{\lceil e_i' \rceil})$	by substitution
$[\![\Delta, \Gamma]\!] \vdash \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{drop}\ \Gamma_i';\ e_i'}\ \} \dashv [\![\Delta]\!]$	SRMATCH

case

$\Delta \mid \Gamma \vdash C\ v_1 \ldots v_n \rightsquigarrow C\ v_1' \ldots v_n'$	given
$\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \vdash_s v_i \rightsquigarrow v_i'$	SCON
$\Gamma_i \ \hat{=}\ (\Gamma - \Gamma_{i+1} - \ldots - \Gamma_n) \cap \mathsf{fv}(v_i)$	SCON
$[\![\Delta, \Gamma_i, \Gamma_{i+1}, \ldots, \Gamma_n]\!] \vdash_s v_i \dashv [\![\Delta, \Gamma_{i+1}, \ldots, \Gamma_n]\!]$	I.H.
$[\![\Delta, \Gamma]\!] \vdash_s C\ v_1 \ldots v_n \dashv [\![\Delta]\!]$	SRCON

□

D.5.4 Precision.

Lemma 22. (*Reachability for erased expressions*)
If $\mathsf{H}_1 \vdash_s e \dashv \mathsf{H}_2$, then $\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H}_1 \mid \lceil e \rceil)$.

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

$\mathsf{H} \vdash_s C\ v_1 \ldots v_n \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash_s v_1 \dashv \mathsf{H}_1 \ldots \mathsf{H}_{n-1} \vdash_s v_n \dashv \mathsf{H}_n$	SRCON
$\mathsf{reach}(\mathsf{H}_{i-1} - \mathsf{H}_i,\ \mathsf{H}_{i-1} \mid \lceil v_i \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H}_{i-1} - \mathsf{H}_i,\ \mathsf{H}_0 \mid \lceil v_i \rceil)$	Lemma 17
$\mathsf{reach}(\mathsf{H}_n - \mathsf{H}_0,\ \mathsf{H}_0 \mid \lceil C\ v_1 \ldots v_n \rceil)$	Follows

case

$\mathsf{H},\ x \mapsto^{n+1} v \vdash_s x \dashv \mathsf{H},\ x \mapsto^n v$	given
$\mathsf{dom}(\mathsf{H},\ x \mapsto^{n+1} v) - \mathsf{dom}(\mathsf{H},\ x \mapsto^n v) = \varnothing$	

case

$\mathsf{H},\ x \mapsto^1 \lambda^{ys} z.\ e \vdash_s x \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash_s ys \dashv \mathsf{H}_1$	SRVARLAM
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil ys \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil \lambda^{ys} z.\ e) \rceil$	by definition
$\mathsf{reach}((\mathsf{H}, x \mapsto^1 \lambda^{ys} z.\ e) - \mathsf{H}_1,\ (\mathsf{H},\ x \mapsto^1 \lambda^{ys} z.\ e) \mid \lceil x \rceil)$	follows

case

$\mathsf{H},\ x \mapsto^1 C\ ys \ \vdash_s x - \ \mid \mathsf{H}_1$	given
$\mathsf{H} \ \vdash_s ys \ \dashv \mathsf{H}_1$	SRVARCON
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil ys \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil C\ ys \rceil)$	by definition
$\mathsf{reach}((\mathsf{H}, x \mapsto^1 C\ ys) - \mathsf{H}_1,\ (\mathsf{H}, x \mapsto^1 C\ ys) \mid \lceil x \rceil)$	follows

case

$\mathsf{H} \ \vdash_s \lambda^{ys}\ x.\ e \ \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash_s ys \ \dashv \mathsf{H}_1$	SRLAM1
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil ys \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil \lambda^{ys} x.\ e) \rceil$	follows

case

$\mathsf{H} \ \vdash_s \lambda^{ys}\ x.\ e \ \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash_s ys \ \dashv \mathsf{H}_1$	SRLAM2
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil ys \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil \lambda^{ys} x.\ \mathsf{drop}\ x;\ e) \rceil$	follows

case

$\mathsf{H} \ \vdash_s e_1\ e_2 \ \dashv \mathsf{H}_2$	given
$\mathsf{H} \ \vdash_s e_1 \ \dashv \mathsf{H}_1$	SRAPP
$\mathsf{H}_1 \ \vdash_s e_2 \ \dashv \mathsf{H}_2$	SRAPP
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil e_1 \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H}_1 \mid \lceil e_2 \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H} \mid \lceil e_2 \rceil)$	Lemma 17
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_2,\ \mathsf{H} \mid \lceil e_1\ e_2 \rceil)$	follows

case

$\mathsf{H} \ \vdash_s \mathsf{val}\ x \ = \ e_1\ ;\ e_2 \ \dashv \mathsf{H}_2$	given
$\mathsf{H} \ \vdash_s e_1 \ \dashv \mathsf{H}_1$	SRBIND1
$\mathsf{H}_1, x \mapsto^1 () \vdash_s e_2 \ \dashv \mathsf{H}_2$	SRBIND1
$x \notin \mathsf{H},\ \mathsf{H}_2$	SRBIND1
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil e_1 \rceil)$	I.H.
$\mathsf{reach}((\mathsf{H}_1, x \mapsto^1 ()) - \mathsf{H}_2,\ (\mathsf{H}_1, x \mapsto^1 ()) \ \mid \lceil e_2 \rceil)$	I.H.
$\mathsf{reach}((\mathsf{H}_1, x \mapsto^1 ()) - \mathsf{H}_2,\ (\mathsf{H}, x \mapsto^1 ()) \ \mid \lceil e_2 \rceil)$	Lemma 17
$\mathsf{dom}(\mathsf{H}_1) \ \subseteq \mathsf{dom}(\mathsf{H}_1, x \mapsto^1 ())$	
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ (\mathsf{H}, x \mapsto^1 ()) \ \mid \lceil e_2 \rceil)$	follows
$x \notin \mathsf{H}$	known
$x \notin \mathsf{dom}(\mathsf{H}) - \mathsf{dom}(\mathsf{H}_2)$	follows
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_2,\ \mathsf{H} \mid \lceil \mathsf{val}\ x \ = \ e_1;\ e_2 \rceil)$	follows

case

$\mathsf{H} \ \vdash_s \mathsf{val}\ x \ = \ e_1\ ;\ \mathsf{drop}\ x;\ e_2 \ \dashv \mathsf{H}_2$	given
$\mathsf{H} \ \vdash_s e_1 \ \dashv \mathsf{H}_1$	SRBIND2
$\mathsf{H}_1 \vdash_s e_2 \ \dashv \mathsf{H}_2$	SRBIND2
$x \notin \mathsf{H}$	SRBIND2
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil e_1 \rceil)$	I.H.
$\mathsf{reach}((\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H}_1 \mid \lceil e_2 \rceil)$	I.H.
$\mathsf{reach}((\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H} \mid \lceil e_2 \rceil)$	Lemma 17
$x \notin \mathsf{H}$	known
$x \notin \mathsf{dom}(\mathsf{H}) - \mathsf{dom}(\mathsf{H}_2)$	follows
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_2,\ \mathsf{H} \mid \lceil \mathsf{val}\ x \ = \ e_1;\ \mathsf{drop}\ x;\ e_2 \rceil)$	follows

case

$\mathsf{H} \vdash_s \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{drop}\ ys_i;\ e_i}\ \} \dashv \mathsf{H}'$	given
$\mathsf{H} \vdash_s x \dashv \mathsf{H}_1$	SRMATCH
$\mathsf{H}_1,\ [\![\mathsf{bv}(p_i)]\!] \vdash_r \mathsf{drop}\ ys_i;\ () \dashv \mathsf{H}_i$	SRMATCH
$\mathsf{H}_i \vdash_s e_i \dashv \mathsf{H}'$	SRMATCH
$\mathsf{bv}(p_i) \notin \mathsf{H}, \mathsf{H}'$	SRMATCH
$ys_i \subseteq \mathsf{fv}(\overline{\lceil e_i \rceil}) \cup \mathsf{bv}(p_i)$	SRMATCH
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid x)$	I.H.
$\mathsf{H}_1, [\![\mathsf{bv}(p_i)]\!] = \mathsf{H}_i +\!\!\!+\ ys_i$	Lemma 18
$\mathsf{reach}((\mathsf{H}_i - \mathsf{H}',\ \mathsf{H}_i \mid \lceil e_i \rceil)$	I.H.
$\mathsf{reach}((\mathsf{H}_i - \mathsf{H}',\ (\mathsf{H}, [\![\mathsf{bv}(p_i)]\!]) \mid \lceil e_i \rceil)$	Lemma 17
$ys_i \subseteq \mathsf{fv}(\overline{\lceil e_i \rceil}) \cup \mathsf{bv}(p_i)$	known
$\mathsf{reach}((\mathsf{H}_i +\!\!\!+ ys_i - \mathsf{H}',\ \mathsf{H} \mid \mathsf{bv}(p_i);\ \overline{\lceil e_i \rceil)}$	follows
$\mathsf{reach}((\mathsf{H}_1, [\![\mathsf{bv}(p_i)]\!] - \mathsf{H}',\ \mathsf{H} \mid \mathsf{bv}(p_i);\ \overline{\lceil e_i \rceil})$	by substitution
$\mathsf{bv}(p_i) \notin \mathsf{H}$	known
$\mathsf{bv}(p_i) \notin \mathsf{dom}(\mathsf{H}) - \mathsf{dom}(\mathsf{H}')$	follows
$\mathsf{reach}(\mathsf{H} - \mathsf{H}',\ \mathsf{H} \mid \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \lceil e_i \rceil}\ \}\ )$	follows

case

$\mathsf{H},\ x \mapsto^n v \vdash_s \mathsf{dup}\ x;\ e \dashv \mathsf{H}_1$	given
$\mathsf{H},\ x \mapsto^{n+1} v \vdash_s e \dashv \mathsf{H}$	SRDUP
$x \in \mathsf{fv}(\lceil e \rceil)$	SRDUP
$\mathsf{reach}((\mathsf{H}, x \mapsto^{n+1} v) - \mathsf{H},\ (\mathsf{H}, x \mapsto^{n+1} v) \mid e)$	I.H.
$\mathsf{reach}((\mathsf{H}, x \mapsto^{n} v) - \mathsf{H},\ (\mathsf{H}, x \mapsto^{n} v) \mid e)$	Follows

□

Proof. (*Of Theorem 4*)

$\varnothing \mid \varnothing \vdash_{\mathsf{s}} e \rightsquigarrow e'$	given
$\varnothing \vdash_r e' \dashv \varnothing$	Lemma 21
$\mathsf{H}_i \vdash_r e_i \dashv \varnothing$	Theorem 7
$e_i \neq \mathsf{E}[\mathsf{dup}\ x;\ e_i']$	given
$\mathsf{H}_i \vdash_s e' \dashv \varnothing$	RRSUB
$\mathsf{reach}(\mathsf{H}_i - \varnothing,\ \mathsf{H}_i \mid \lceil e \rceil)$	Lemma 22
all $y \in \mathsf{dom}(\mathsf{H}_i)$, $\mathsf{reach}(y,\ \mathsf{H}_i \mid e_i)$	follows

□