Reference Counting with Frame Limited Reuse

Microsoft Technical Report, MSR-TR-2021-30, Mar 15, 2022 (v2).

ANTON LORENZEN, University of Bonn, Germany
DAAN LEIJEN, Microsoft Research, USA

The recently introduced *Perceus* algorithm can automatically insert reference count instructions such that the resulting (cycle-free) program is *garbage free*: objects are freed at the very moment they can no longer be referenced. An important extension is reuse analysis. This optimization pairs objects of known size with fresh allocations of the same size and tries to reuse the object in-place at runtime if it happens to be unique. Unfortunately, current implementations of reuse analysis are fragile with respect to small program transformations, or can cause an arbitrary increase in the peak heap usage. We present a novel *drop-guided* reuse algorithm that is simpler and more robust than previous approaches. Moreover, we generalize the linear resource calculus to precisely characterize garbage-free and frame-limited evaluations. On each function call, a frame-limited evaluation may hold on to memory longer if the size is bounded by a constant factor. Using this framework we show that our drop-guided reuse *is* frame-limited and find that an implementation of our new reuse approach in Koka can provide significant speedups.

1 INTRODUCTION

Reference counting (Collins, 1960) is a technique for automatic memory management where each allocated object stores the number of references that point to it. Reinking, Xie, de Moura, and Leijen (2021) describe the *Perceus* algorithm for automatically inserting reference count instructions such that the resulting (cycle-free) program is *garbage free*: objects are freed at the very moment they can no longer be referenced. Even though the Perceus algorithm itself needs an internal calculus with explicit control flow, the authors apply this work in the context of the Koka language (2021) which supports full algebraic effect handlers. These can be used to define various kinds of implicit control flow, including features like exceptions, async/await, and backtracking (Xie and Leijen, 2021; Leijen, 2017; Plotkin and Pretnar, 2009; Plotkin and Power, 2003).

An important extension is *reuse analysis*. This optimization pairs objects of known size with fresh allocations of the same size and tries to reuse the object in-place at runtime if it happens to be unique. This was first described in earlier work by Ullrich and de Moura (2019) in the context of the Lean theorem prover (Moura and Ullrich, 2021). Unfortunately, the published algorithms for reuse analysis all have various weaknesses; for example, they are fragile with respect to small program transformations, where inlining or rearranging expressions can cause reuse analysis to fail unexpectedly.

Moreover, while Perceus itself is garbage-free, reuse analysis does not have this property. By construction, it holds on to memory that is to be reused later, which can lead to an increased peak memory usage. We find the maximum increase is not just a constant factor but can be much larger, and is generally not *safe for space* (Paraskevopoulou and Appel, 2019; Appel, 1991).

In this work we improve upon this with a novel reuse algorithm and a formal framework for reasoning about heap bounds that can be applied to general transformations (including reuse analysis). In particular:

- We define a new approach to reuse called *drop guided reuse* (Section 3.2). In contrast to earlier techniques we perform the reuse *after* Perceus has inserted reference count instructions. This both simplifies the analysis and makes it more robust with respect to small program transformations. We formalize drop-guided reuse generally in the form of declarative derivation rules where we can discuss clearly the various choices an algorithm can make (Section 5.4).

- We illustrate this with a practical example in Section 2.2.1 where we see significantly better reuse for red-black tree balanced insertion in comparison with the previous algorithms. In combination with the tail-recursion-modulo-cons (TRMC) optimization, we show how for red-black tree insertion we get a highly optimized code path reusing a (unique) tree in-place while using minimal stack space. Our straightforward and purely functional implementation is about 19% faster than the manually optimized in-place mutating red-black tree implementation in the C++ STL library (`std::map`) (Section 6).
- We reformulate the original linear resource calculus λ^1 (Reinking, Xie et al., 2021) in a normalized form (λ^{1n}) where we can reason precisely about reuse. As for λ^1, the declarative derivation rules for the λ^{1n} calculus are non-deterministic and can derive many programs with reference counting instructions that are all correct but may differ in their memory consumption. However, in the normalized reformulation, it now is possible to add a single logical side condition ($\star$) to the LET rule that captures various important variants concisely (Section 5.3):
  - If the ($\star$) condition is unrestricted, the resulting programs are *sound* – that is, the reference counting is correct and the final heap contains no garbage.
  - By restricting ($\star$) in a particular way, we can show that all derived programs are *garbage-free* where at every allocating evaluation step there is no garbage (and all programs resulting from the Perceus algorithm fall in this class).
  - Finally, we can weaken the ($\star$) condition of the garbage-free system to also allow derivations that we call *frame-limited*, where every function call uses at most a constant factor c more memory.
- Transformations like reuse and borrowing no longer have the garbage-free property as they hold on to some memory for a bit longer (the cell to reuse, or the data that is borrowed). With the new formalization, we can now show that some of these transformations are still *frame-limited*, and we prove that our new *drop-guided* reuse analysis *is* frame-limited (Section 5.4). In contrast, we show that some previous reuse algorithms and unrestricted borrow inference (Ullrich and de Moura, 2019) are *not* frame-limited transformations (and we argue such transformations should therefore be avoided in practice).
- Building on robust reuse analysis, we can often express imperative style algorithms in a functional way. Such algorithms are called *Functional But In-Place* (FBIP). In Section 7 we use this technique to create a faster version of the parallel `binarytrees` benchmark. We then generalize this approach to visitor data types (as a *derivative* of the original data type) to derive a novel FBIP algorithm for red-black tree balanced insertion that further improves upon the standard Okisaki style implementation (Okasaki, 1999a).

2 OVERVIEW AND BACKGROUND

We start with a short overview of background material and related work; in particular the Koka language, Perceus-style reference counting, and reuse analysis. All our examples use the Koka language (Leijen, 2021, 2017, 2014; Xie and Leijen, 2021) – a strongly typed functional language with effect handlers which tracks (side) effects in the type of every function. 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

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.

2.1 Perceus

By starting from a language with strong static guarantees (like Koka), the Perceus algorithm (Reinking, Xie et al., 2021) can insert optimized reference count instructions during compilation. Note though it still needs separate mechanisms to address cyclic data and mitigate the impact of thread shared reference counts – we refer to the Perceus paper for a in-depth discussion of these.

The main attribute that sets Perceus apart from most automatic memory management systems is that it is *garbage-free*: for a cycle-free program, an object is freed as soon as no more references remain. Consider for example the following function:

```
fun main()
  val xs = list(1,1000000)  // allocate a large list
  val ys = map(xs,inc)      // increment each element
  println(ys)
```

Many reference count systems would drop the references to `xs` and `ys` based on the lexical scope; for example:

```
fun main()
  val xs = list(1,1000000)
  val ys = map(xs,inc)
  println(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 for example used by a 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`) (Yarantsev, 2020). It is not required by the semantics, but Swift typically emits code like this as well (Gallagher, 2016).

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. Moreover, at the end of the scope, a long cascading chain of drop operations happens for each element in both lists.

2.1.1 Ownership. 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 `main` as all local variables are *consumed* by other functions, while the `map` and `print` functions drop the list elements as they go. Let's take a look at what reference count instructions Perceus generates for the `map` function:

```
fun map(xs : list⟨a⟩, f : a -> e b) : e list⟨b⟩
  match xs
    Cons(x,xx) ->
      dup(x); dup(xx); drop(xs)
      Cons( dup(f)(x), map(xx,f))
    Nil ->
      drop(xs); drop(f)
      Nil
```

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.

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 constructed.

2.2 Reuse

Reuse analysis (Reinking, Xie et al., 2021; Ullrich and de Moura, 2019) is an optimization that takes advantage of precise reference counts to try to reuse objects in-place. We can pair objects of known size with same sized allocated constructors and try to reuse these in-place at runtime. Reuse analysis rewrites `map` into:

```
fun map(xs : list⟨a⟩, f : a -> e b) : e list⟨b⟩
  match xs
    Cons(x,xx) ->
      dup(x); dup(xx); val r = dropru(xs)
      Cons@r( dup(f)(x), map(xx,f))
```

The *reuse token* `r` becomes the address of the `Cons` cell `xs` if `xs` happens to be unique, and `NULL` otherwise. The `Cons@r` allocation reuses `xs` in-place if `r` is non `NULL`, and otherwise allocates a fresh `Cons` cell. In case we map over a unique list, the list elements are updated in-place. A further rewriting technique called *drop specialization* can further optimize this by inlining the `dropru` operation and simplifying such that no reference count operations are necessary in the case that the list is unique:

```
fun map(xs : list⟨a⟩, f : a -> e b) : e list⟨b⟩
  match xs
    Cons(x,xx) ->
      val r = if unique(xs) then &xs    // reuse the address of xs directly
              else dup(x); dup(xx); decref(xs); NULL
      Cons@r( dup(f)(x), map(xx,f))
    ...
```

This is an important optimization in practice and our new reuse algorithm is compatible with it, but for clarity we generally leave it out in the following examples.

2.2.1 Balanced Trees. Reinking, Xie et al. (2021) provide an appealing example of the effectiveness of reuse analysis using balanced insertion in red-black trees (Guibas and Sedgewick, 1978). Figure 1 shows the implementation in Koka based on Okasaki's algorithm (Okasaki, 1999a). A red-black tree has the invariant that the number of black nodes from the root to any of the leaves are 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 are maintained by rebalancing the nodes when needed.

If we look closely at all the `match` branches in Figure 1, (and assume that the `lbal` and `rbal` functions get inlined), then we can see that we always either match one `Node` and allocate one,

```
type color { R; B }

type tree⟨a⟩
  Node( clr:color, l:tree⟨a⟩, key:int, value:a, r:tree⟨a⟩ )
  Leaf

fun is-red(t : tree⟨a⟩) : bool
  match t
    Node(R,_,_,_,_) -> True
    _               -> False

fun lbal(l :tree⟨a⟩, k : int, v : a, r : tree⟨a⟩) : tree⟨a⟩
  match l
    Node(_, Node(R, lx, kx, vx, rx), ky, vy, ry) ->
      Node(R, Node(B,lx,kx,vx,rx), ky, vy, Node(B,ry,k,v,r))
    Node(_, ly, ky, vy, Node(R, lx, kx, vx, rx)) ->
      Node(R, Node(B,ly,ky,vy,lx), kx, vx, Node(B,rx,k,v,r))
    Node(_, lx, kx, vx, rx) ->
      Node(B, Node(R,lx,kx,vx,rx), k, v, r)
    Leaf -> Leaf

fun rbal(l : tree⟨a⟩, k : int, v : a, r : tree⟨a⟩) : tree⟨a⟩
  ...

fun ins(t : tree⟨a⟩, k : int, v : a) : tree⟨a⟩
  match t
    Leaf -> Node(R, Leaf, k, v, Leaf)
    Node(B, l, kx, vx, r) ->
      if k < kx then
        if is-red(l) then lbal(ins(l,k,v), kx, vx, r)
                     else Node(B, ins(l,k,v), kx, vx, r)
      elif k > kx then
        if is-red(r) then rbal(l, kx, vx, ins(r,k,v))
                     else Node(B, l, kx, vx, ins(r,k,v))
      else Node(B, l, k, v, r)
    Node(R, l, kx, vx, r) ->
      if k < kx then Node(R, ins(l,k,v), kx, vx, r)
      elif k > kx then Node(R, l, kx, vx, ins(r,k,v))
      else Node(R, l, k, v, r)
```

Fig. 1. Balanced red-black tree insertion

or we match three nodes deep and allocate three (when rebalancing). In the case that the tree is unique, the reuse analysis reuses every `Node` along the spine without doing any further allocations! Moreover, if we use the tree *persistently* (Okasaki, 1999b) where the tree is shared (or has shared subtrees), it adapts to copying exactly the shared spine of the tree (and no more).

3 DROP-GUIDED REUSE

Previous work has shown that reuse analysis can be very effective, and has been implemented in both the Koka and Lean languages. Unfortunately, it turns out that both previously published algorithms for reuse analysis are flawed: the Koka algorithm, which we call algorithm K (Reinking, Xie et al., 2021), is fragile with respect to small program tranformations, where rearranging expressions can cause reuse analysis to fail unexpectedly. The Lean algorithm, called algorithm D (Ullrich and de Moura, 2019 (Fig 3)), is more robust but can lead to an arbitrary increase in peak memory usage. In this section we look at each algorithm, and propose a new approach, called *drop-guided reuse*, that improves upon both the previous techniques.

3.1 Problems with Reuse

Both Reinking, Xie et al. (2021) and Ullrich and de Moura (2019) describe reuse algorithms as a pass *before* the main Perceus algorithm. The reason they chose this approach is two-fold: the `dropru` function can be seen as consuming its argument and thus needs no special treatment from Perceus, and similarly, a reuse token can be deallocated by Perceus if it is not used (for example if the constructor is only allocated in one branch of a nested match-statement but not another). However, in practice we observed that this can lead to situations where the reuse is not optimal.

Algorithm K as implemented in the Koka compiler tries to reuse at the start of a branch whenever a matching constructor size can be found in the branch body. That seems reasonable at first, but consider the following example (A):

```
match x
  Just(_) -> // x is still live here
    match y
      0 -> x
      _ -> Just(y)
```

In this case, the `Just(y)` matches with the deconstructed `x`, where reuse analysis will try to reuse `x`, and the generated code becomes:

```
match x
  Just(_) ->
    val r = dropru(dup(x))
    match y
      0 -> drop(y); drop(r); x
      _ -> drop(x); Just@r(y)
```

Unfortunately, since `x` is still live in the scope it prevents any reuse at runtime due to the inserted `dup` operation, and `r` will always be `NULL` at runtime.

An obvious way to improve on this, is by pushing down reuse operations into branches behind the last use of an object. This is the approach used by algorithm D, where example (A) can reuse `x` now effectively:

```
match x
  Just(_) ->
    match y
      0 -> drop(y); x
      _ -> val r = dropru(dup(x)); Just@r(y)
```

This approach has a different weakness though, and can lead to an arbitrary increase in peak memory usage. Consider the following example (B):

```
match xs
  Cons(_,_) ->
    val y = f(xs)
    Cons(y,Nil)
```

Using algorithm D, a reuse is inserted right after the call to `f`, which leads to Perceus (running afterwards) inserting a `dup` on the `xs` parameter:

```
match xs
  Cons(_,_) ->
    val y = f(dup(xs))
    val r = dropru(xs)
    Cons@r(y,Nil)
```

Even though the `Cons` cell of `xs` is actually available for reuse, it now also holds on to the full `xs` list during evaluation of `f`. This not only means that we may use `sizeof(xs)` more memory than necessary, but also that any reuse by `f` of `xs` is prevented was well (as `xs` is now certainly not unique).

3.1.1 Reuse in Balanced Trees. The previous patterns actually occur regularly in practice. In fact, both algorithm D and K also fail to effectively reuse even under small rewrites the balanced tree insertion example. Consider again the code in Figure 1, and let's focus on the second branch in the `ins` function:

```
match t
  Node(B, l, kx, vx, r) ->
    if k < kx then
```

```
    if is-red(l) then lbal(ins(l,k,v), kx, vx, r)
                 else Node(B, ins(l,k,v), kx, vx, r))
```

With both reuse algorithms, the `Node` allocation in the `else` branch will reuse the outer matched node `t`. It gets more interesting though, if both `lbal` *and* `is-red` get inlined (and simplified); we focus just on the `then` case and the first branch of `lbal` here:

```
match t
  Node(B, l, kx, vx, r) ->
    if k < kx then
      match l
        Node(R,_,_,_,_) ->
          val t2 = ins(l,k,v)
          match t2
            Node(_, Node(R, ...), ...) ->
              Node(R, Node(B, ...), ..., Node(B, ...))
```

Due to the inlined `is-red` function, we now get a situation where both algorithm D and K fall short. This is exactly like example (B) in Section 3.1, and where algorithm K will try to reuse `l` even though it is still used later on, leading to:

```
match l
  Node(R,_,_,_,_) ->
    val ru2 = dropru(dup(l))
    val u = ins(l,k,v)
```

where $\texttt{ru}_2$ is never available for reuse at runtime. Algorithm D does not fare much better either as it holds on to `l` preventing the recursive `ins` from reusing the tree:

```
match l
  Node(R,_,_,_,_) ->
    val t2 = ins(dup(l),k,v)
    val ru2 = dropreuse(l)
```

As shown, both published algorithms are quite fragile with respect to small program tranformations.

3.2 Drop Guided Reuse

We believe the essential weakness of both previous algorithms is that they fail to take liveness into account. As such, we propose to perform reuse analysis *after* doing Perceus dup-drop insertion.

Indeed, Perceus already performs precise liveness analysis and it has been shown it inserts optimal dup/drop operations in the sense that the resulting program is garbage-free. This means that the `drop` operations signify precisely when a cell may become available for reuse – and this is exactly the point where we should rewrite the `drop` to a `dropru` if there is any potential for reuse. The call to `dropru` cannot hold on to objects any longer than necessary since by the garbage-free property an object must be live until directly before it is dropped.

With *drop-guided* reuse analysis, we keep track of the currently known sizes of each variable (updated at each branch pattern) and if we encounter a `drop` we can statically determine if it can pair with a later allocation of the same size. Looking at our earlier example (A) from the previous section, Perceus generates first:

```
match x
  Just(_) ->
    match y
      0 -> drop(y); x
      _ -> drop(x); Just(y)
```

The *drop-guided* reuse analysis can now rewrite the `drop(x)` into a `dropru(x)` as we know that the size of `x` matches the size of the following `Just` allocation:

```
match x
  Just(_) ->
    match y
      0 -> drop(y); x
      _ -> val r = dropru(x); Just@r(y)
```

For the other example (B) in the previous section, no `drop` operations are generated in the first place, and no `dropru` is inserted either – perfect!

A drawback of the new approach is that we need to explicitly free newly created reuse tokens if these happen to be unused in some branch, and we can no longer rely on Perceus doing this for us. It may seem we need to perform another mini Perceus pass to address this, but it can be done in a simpler way: since reuse tokens are only generated in a specific way, we can show they are only used either never or once, and never captured under a lambda or passed as an argument. As such, it is suffices to locally check at each branch of a match expression if a given reuse token `r` occurs in this branch and insert a `free(r)` instruction if this is not the case. In the new Koka implementation we combine this in one pass with the reuse analysis.

Finally, with balanced tree insertion, none of the earlier problems with algorithm D and K occur. The drop-guided approach is robust and leads to optimal reuse:

```
match t
  Node(B, l, kb, vb, r) ->
    dup(l); dup(kb); dup(vb); dup(r)
    val ru = dropreuse(t)
    if k < kb then
      match l
        Node(R,_,_,_,_) ->
          val t2 = ins(l,k,v)
          match t2
            Node(_, l2 as Node(R, ...), ..., r2) ->
              val ru2 = dropreuse(t2)
              val ru3 = dropreuse(l2)
              Node@ru(R, Node@ru2(B, ...), ..., Node@ru3(B, ...))
```

Again, the reuse analysis will reuse every `Node` along the spine without doing any further allocations if the tree is unique. As we show in the benchmarking section, with the new improved reuse analysis the performance on unique trees rivals that of the manually optimized in-place mutating red-black tree implementation in the C++ STL library (`std::map`).

3.2.1 TRMC: Tail-Recursion-Modulo-Cons. We can make the red-black tree insertion a bit faster still with *tail-recursion-modulo-cons* (TRMC) optimization. Usually, with *tail-recursion* any functions that call themselves recursively in a tail position are transformed into a loop instead (using no extra stack space). With TRMC, such function can make its tail call inside any tail-position *expression* consisting of just constructors and non-allocating total expressions. For example, `map` is such function where the recursion in `Cons(f(x), map(f,xx))` is transformed into a loop. This is done by pre-allocating the `Cons` node ahead of the recursive call with a *hole* in the tail field, which is later assigned by the recursive call.

TRMC interacts well with reuse analysis as often the recursive call is inside a constructor that is reused. In the `map` function for example, this will result in traversing the list in a tight loop while updating each element in-place when the list `xs` happens to be unique.

For red-black tree rebalancing, we can see in Figure 1 that there are four TRMC recursive `ins` calls and only in the `rbal`/`lbal` cases this does not hold. When we study the generated C code the final result is quite sophisticated: there is an outer TRMC loop for the four TRMC `ins` calls, but it is interspersed at runtime with the two actual recursive `ins` calls. Moreover, due to reuse, the code updates unique nodes in place when rebalancing the spine, but adapts to copying for shared subtrees. Overall this is very efficient code that would be difficult to write directly by hand.

4 REASONING ABOUT SPACE

As we have seen, liveness analysis helps us avoid the problems of algorithm K, but have we also avoided the problem of algorithm D of using too much space? Indeed, since reuse analysis keeps heap cells alive until they can be reused, it means that no reuse analysis can preserve the garbage-free property! Is there a way to still characterize the space usage of such transformations that is more restrictive than allowing arbitrary increases, but also more permissive than garbage-free?

4.1 Frame-Limited Transformations

Drop guided reuse analysis can, at any evaluation step, hold on to a single cell per reuse token `r` that was created by `dropru`, but not used at a constructor yet. Since any function can only contain a constant number of `dropru` calls, one might expect the total overhead to be constant as well, which would make drop guided reuse *safe for space* (Paraskevopoulou and Appel, 2019; Appel, 1991), in the sense that the maximum peak memory increase is bounded by a constant. However, before a reuse token is used there might be a recursive call. Consider the `map` function we first viewed:

```
val r = dropru(xs)
Cons@r( dup(f)(x), map(xx,f))
```

Here, `r` is live during the recursive call and so reuse analysis can hold on to as many `Cons` cells as either the list is long or the stack allows. In other words, we can only hope to bound the extra memory needed for reuse analysis by a constant factor times the current number of stack frames – we call this *frame-limited*. We formalize this notion in the next section, and formally prove in Section 5.4 that our new drop guided reuse *is* a frame-limited transformation. In contrast, as we showed in Section 3.1, the reuse algorithm D (Ullrich and de Moura, 2019) is not frame limited and can lead to an arbitrary increase in memory usage.

Even though weaker than being garbage-free or safe for space, we argue that frame-limited transformations are still good in practice. First of all, programmers are already aware of recursion and take steps to avoid unbounded recursion. Secondly, in practice the stack size is usually already bounded – in such case, that makes the frame-limited bound constant (and thus safe for space).

Note that in practice, backend optimizations like tail-recursion may optimize stack frames away. However, we formalize frame-limited in terms of the size of the evaluation context (i.e. the recursion depth) instead of actual stack frames and thus the bound still applies.

4.2 Borrowing

Another example of a transformation that does not preserve the garbage-free property is *borrowing* (Ullrich and de Moura, 2019): even though the first example in Section 2.1 argues that arguments should be passed owned to the callee, this is not always optimal – sometimes it is better to pass arguments as *borrowed* instead. Consider converting a list into an unbalanced binary tree:

```
fun make-tree( xs : list⟨a⟩ ) : tree⟨a⟩
  match xs
    Cons(x, xx) -> Node( R, Leaf, 0, x, make-tree(xx) )
    Nil -> Leaf
```

Perceus inserts `dup(x); dup(xx); drop(xs);` in the `Cons` branch, which causes some overhead, especially since there are no reuse opportunities. When we annotate a parameter like `xs` to be borrowed (as `^xs`), the caller keeps ownership of the parameter. As a result, no reference count operations need to be performed at all for the `xs` parameter in our example. Note that borrow annotations are strictly a performance hint, and do not change semantics or whether a program is well-typed (in contrast to the notion of borrowing in a language like Rust for example).

Borrow annotations are not always beneficial though: if `xs` happens to be unique, we allocate the tree while the full `xs` list is still live – exactly the situation we wanted to avoid in Section 2.1. In general, borrowing can increase the memory usage of a program by an arbitrary amount and it is generally *not* frame-limited. Ullrich and de Moura (2019) describe a borrow inference algorithm that marks `xs` automatically as borrowed. However, given that this is not safe for space, we argue that automatic borrow inference should be further restricted to guarantee it is at least frame-limited. Therefore, Koka currently has no automatic borrow inference and generally only uses borrowing for built-in primitives (like (big) integer operations).

5 FORMALIZATION

$$
\begin{array}{lcl}
e & ::= & v \\
  & \mid & e\ e \\
  & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
  & \mid & \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
\qquad
\begin{array}{lcl}
v & ::= & x \\
  & \mid & \lambda x.\ e \\
  & \mid & C\ \overline{v} \\
p & ::= & C\ \overline{x}
\end{array}
$$

Semantics:

$$
\begin{array}{lll}
\mathsf{E} ::= \Box \mid \mathsf{E}\ e \mid v\ \mathsf{E} \\
\quad \mid \ \mathsf{let}\ x = \mathsf{E}\ \mathsf{in}\ e \\
\quad \mid \ \mathsf{match}\ \mathsf{E}\ \{\ \overline{p_i \to e_i}\ \}
\end{array}
\qquad
\frac{e \longrightarrow e'}{\mathsf{E}[e] \longmapsto \mathsf{E}[e']}\ [\textsc{eval}]
$$

Small step transitions:

$$
\begin{array}{llcl}
(app) & (\lambda x.\ e)\ v & \longrightarrow & e[x{:=}v] \\
(let) & \mathsf{let}\ x = v\ \mathsf{in}\ e & \longrightarrow & e[x{:=}v] \\
(match) & \mathsf{match}\ (C\ \overline{v})\ \{\overline{p_i \to e_i}\} & \longrightarrow & e_i[\overline{x}{:=}\overline{v}] \\
 & & & \text{where } p_i\ =\ C\ \overline{x}
\end{array}
$$

Fig. 2. Syntax and semantics of λ^1

We formalize our results using the linear resource calculus λ^1 as given by Reinking, Xie et al. (2021) (see figure 2). This is essentially just lambda calculus extended with let bindings and pattern matching. We assume that the patterns p_i in a match are all distinct. The semantics for λ^1 is standard using strict evaluation contexts E (Wright and Felleisen, 1994). The evaluation contexts uniquely determine where to apply an evaluation step using the EVAL rule. As such, evaluation contexts neatly abstract from the usual implementation context of a stack and program counter. The small step evaluation rules perform function application (app), let-binding (let), and pattern matching $(match)$.

5.1 Heap Semantics

To reason precisely about reference counting, we need a semantics with an explicit heap. Reinking, Xie et al. (2021) define a heap semantics directly over λ^1. However, since we aim to reason precisely about the space behavior, this is not quite sufficient for our case as we need to be more explicit about sharing and evaluation order. We therefore translate any expression e into *normalized form* $\lfloor e \rfloor$, as defined in Figure 3, where all arguments become variables instead of values (Flanagan et al., 1993).

Figure 4 defines the syntax of the *normalized linear resource calculus* λ^{1n} where all arguments are now variables. Moreover the 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, while $r \leftarrow \mathsf{dropru}$ is used for reuse. Also, every lambda $\lambda_{\overline{z}} x.e$ is annotated with its free variables $\overline{z}$ which becomes important during evaluation.

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 $\mathsf{bv}(p)$.

Using our normalized calculus, Figure 5 defines the semantics in terms of a reference counted heap, where sharing of values is 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 $\lambda_{\overline{z}} x.\ e$ to clarify that these are essentially *closures* holding an environment $\overline{z}$ and a

$$
\begin{array}{lcl}
\lfloor x \rfloor & = & x \\
\lfloor \lambda x.\ e \rfloor & = & \lambda x.\ \lfloor e \rfloor \\
\lfloor e\ e' \rfloor & = & \mathsf{let}\ f = \lfloor e \rfloor\ \mathsf{in}\ \mathsf{let}\ x = \lfloor e' \rfloor\ \mathsf{in}\ f\ x \\
\lfloor C\ v_1 \ldots v_n \rfloor & = & \mathsf{let}\ x_1 = \lfloor v_1 \rfloor\ \mathsf{in} \ldots \mathsf{let}\ x_n = \lfloor v_n \rfloor\ \mathsf{in}\ C\ x_1 \ldots x_n \\
\lfloor \mathsf{match}\ e\ \{\ \overline{p_i \to e_i}\ \} \rfloor & = & \mathsf{let}\ x = \lfloor e \rfloor\ \mathsf{in}\ \mathsf{match}\ x\ \{\ \overline{p_i \to \lfloor e_i \rfloor}\ \} \\
\lfloor \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 \rfloor & = & \mathsf{let}\ x = \lfloor e_1 \rfloor\ \mathsf{in}\ \lfloor e_2 \rfloor
\end{array}
$$

Fig. 3. Normalization. All f and x are fresh

$$
\begin{array}{lcllcl}
e & ::= & v & v & ::= & x \\
 & \mid & e\ x & & \mid & \lambda_{\overline{z}} x.\ e \\
 & \mid & \mathsf{let}\ x\ =\ e\ \mathsf{in}\ e & & \mid & C\ \overline{x} \\
 & \mid & \mathsf{match}\ x\ \{\ \overline{p_i \to e_i}\ \} & & & \\
 & \mid & \mathsf{dup}\ x;\ e & p & ::= & C\ \overline{x} \\
 & \mid & \mathsf{drop}\ x;\ e & & & \\
 & \mid & r \leftarrow \mathsf{dropru}\ x;\ e & \Delta,\ \Gamma & ::= & \varnothing\ \mid \Delta \cup x
\end{array}
$$

$$\lambda x.e\ \doteq\ \lambda_{\overline{z}} x.e \quad (\overline{z}\ =\ \mathsf{fv}(e))$$

Fig. 4. The normalized linear resource calculus λ^{1n}.

code pointer $\lambda x.\ e$. Note that it is important that the environment $\overline{z}$ is a multi-set. After the initial translation, $\overline{z}$ 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 (shown in Figure 6). First, the (app_r) rule inserts dup to duplicate the free closure variables $\overline{z}$, as these are owned in rule LAM. It then drops the reference to the closure itself.

A difference between (app_r) and $(match_r)$ is that for applications the free variables $\overline{z}$ are dynamic and thus the duplication must be done at runtime. In contrast, a match knows the the bound variables in a pattern statically and we therefore generate the required dup and drop operations statically during elaboration for each branch (as shown in Figure 6) – this is essential as that enables the further static optimizations of dup/drop pairs and reuse analysis.

We discuss the reuse evaluation rules later in Section 5.4.

5.2 Dup-Drop Insertion in λ^{1n}

Figure 6 defines the logical derivation rules over λ^{1n} for inserting reference count instructions such that the resulting expression can be soundly evaluated by the heap semantics of Figure 5. The derivation $\Delta \mid \Gamma \vdash e \ \rightsquigarrow\ e'$ in Figure 6 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*.

The key idea 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.

The rules are close to the derivation rules by Reinking, Xie et al. (2021) but differ in important details. In particular, by using a normalized form we only split the owned environment Γ in the LET rule, and no longer in the APP and CON rules which are now much simpler. This in turn allows us to parameterize the system by a single side condition ($\star$) that allows us to concisely capture *garbage-free*, *frame-limited*, and *sound* transformations as shown in Section 5.3.

$$\mathsf{H} :\ x \mapsto (\mathbb{N}^{+}, v) \qquad\qquad \dfrac{\mathsf{H} \mid e \longrightarrow_r \mathsf{H}' \mid e'}{\mathsf{H} \mid \mathsf{E}[e] \longmapsto_r \mathsf{H}' \mid \mathsf{E}[e']}\ [\textsc{eval}]$$

$$\mathsf{E} ::= \square \mid \mathsf{E}\ x \mid \mathsf{let}\ x = \mathsf{E}\ \mathsf{in}\ e$$

$$\begin{array}{llcll}
(lam_r) & \mathsf{H} \mid \lambda_{\overline{z}}\ x.\ e & \longrightarrow_r & \mathsf{H}, f \mapsto^{1} \lambda_{\overline{z}}\ x.\ e \quad \mid f & \text{fresh } f \\
(con_r) & \mathsf{H} \mid C\ x_1 \ldots x_n & \longrightarrow_r & \mathsf{H}, z \mapsto^{1} C\ x_1 \ldots x_n \quad \mid z & \text{fresh } z \\
(app_r) & \mathsf{H} \mid f\ y & \longrightarrow_r & \mathsf{H} \mid \mathsf{dup}\ \overline{z};\ \mathsf{drop}\ f;\ e[x{:=}y] & (f \mapsto^{n} \lambda_{\overline{z}}\ x.\ e) \in \mathsf{H} \\
(match_r) & \mathsf{H} \mid \mathsf{match}\ y\ \{\overline{p_i \to e_i}\} & \longrightarrow_r & \mathsf{H} \mid e_i[\overline{x}{:=}\overline{z}] & \text{with } p_i = C\ \overline{x} \text{ and } (y \mapsto^{n} C\ \overline{z}) \in \mathsf{H} \\
(let_r) & \mathsf{H} \mid \mathsf{let}\ x = z\ \mathsf{in}\ e & \longrightarrow_r & \mathsf{H} \mid e[x{:=}z] &
\end{array}$$

$$\begin{array}{lllcll}
(dup_r) & \mathsf{H}, x \mapsto^{n} v & \mid \mathsf{dup}\ x;\ e & \longrightarrow_r & \mathsf{H}, x \mapsto^{n+1} v \quad \mid e & \\
(drop_r) & \mathsf{H}, x \mapsto^{n+1} v & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H}, x \mapsto^{n} v \quad \mid e & \text{if } n \geqslant 1 \\
(dlam_r) & \mathsf{H}, x \mapsto^{1} \lambda_{\overline{x}} y.e' & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H} \mid \mathsf{drop}\ \overline{x};\ e & \\
(dcon_r) & \mathsf{H}, x \mapsto^{1} C\ \overline{x} & \mid \mathsf{drop}\ x;\ e & \longrightarrow_r & \mathsf{H} \mid \mathsf{drop}\ \overline{x};\ e &
\end{array}$$

Extension with reuse (with fresh $z, \overline{z}$):

$$\begin{array}{lllcll}
(drop_{ru}) & \mathsf{H}, x \mapsto^{n+1} v & \mid r \leftarrow \mathsf{dropru}\ x;\ e & \longrightarrow_r & \mathsf{H}, x \mapsto^{n} v, z \mapsto^{1} () \mid e[r{:=}z] & \text{if } n \geqslant 1 \\
(dlam_{ru}) & \mathsf{H}, x \mapsto^{1} \lambda_{\overline{x}} y.e' & \mid r \leftarrow \mathsf{dropru}\ x;\ e & \longrightarrow_r & \mathsf{H}, z \mapsto^{1} () \mid \mathsf{drop}\ \overline{x};\ e[r{:=}z] & \\
(dcon_{ru}) & \mathsf{H}, x \mapsto^{1} C\ \overline{x} & \mid r \leftarrow \mathsf{dropru}\ x;\ e & \longrightarrow_r & \mathsf{H}, \overline{z} \mapsto^{1} (), z \mapsto^{1} C\ \overline{z} \mid \mathsf{drop}\ \overline{x};\ e[r{:=}z] &
\end{array}$$

Fig. 5. Reference-counted heap semantics for λ^{1n}.

$$\underset{\uparrow}{\Delta} \mid \underset{\uparrow}{\Gamma} \vdash \underset{\uparrow}{e} \rightsquigarrow \underset{\downarrow}{e'} \qquad (\uparrow \text{ is input}, \downarrow \text{ is output})$$

Δ and Γ are multisets of the borrowed and owned variables in scope

$$\dfrac{}{\Delta \mid x \vdash x \rightsquigarrow x}\ [\textsc{var}] \qquad\qquad \dfrac{}{\Delta \mid \overline{x} \vdash C\ \overline{x} \rightsquigarrow C\ \overline{x}}\ [\textsc{con}]$$

$$\dfrac{\Delta, x \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \Gamma, x \vdash e\ x \rightsquigarrow e'\ x}\ [\textsc{app}] \qquad\qquad \dfrac{\varnothing \mid \Gamma, x \vdash e \rightsquigarrow e' \quad \Gamma = \mathsf{fv}(\lambda x.\ e)}{\Delta \mid \Gamma \vdash \lambda x.\ e \rightsquigarrow \lambda_{\Gamma}\ x.\ e'}\ [\textsc{lam}]$$

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

$$\dfrac{\Delta \mid \Gamma, r \vdash e \rightsquigarrow e' \quad \text{fresh } r}{\Delta \mid \Gamma, x \vdash e \rightsquigarrow r \leftarrow \mathsf{dropru}\ x;\ e'}\ [\textsc{dropru}]$$

$$\dfrac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \quad \Delta \mid \Gamma_2, x \vdash e_2 \rightsquigarrow e_2' \quad (\star)}{\Delta \mid \Gamma_1, \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'}\ [\textsc{let}]$$

$$\dfrac{\Delta \mid \Gamma, \overline{z_i} \vdash e_i \rightsquigarrow e_i' \quad p_i = C_i\ \overline{z_i} \quad x \in \Delta, \Gamma}{\Delta \mid \Gamma \vdash \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \rightsquigarrow \mathsf{match}\ x\ \{\ \overline{p_i \mapsto \mathsf{dup}(\overline{z_i});\ e_i'}\ \}}\ [\textsc{match}]$$

Fig. 6. Logical derivation rules of λ^{1n}. The rules are parameterized by the $(\star)$ condition on LET. We write $\vdash_{\mathsf{GF}}$ for garbage-free derivations, and use $\vdash_{\mathsf{FL}}$ for derivations that are frame-limited.

The LET 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 (Wadler, 1990, pg.14) where a linear type can be "borrowed" as a regular type during evaluation of a binding.

The LAM rule is interesting as it essentially derives the body of the lambda independently. The premise $\Gamma = \mathsf{fv}(\lambda x.e)$ 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 Γ, as $\lambda_\Gamma x.\ e$, 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.

Another important difference from earlier work is that the MATCH rule now statically generates dup instructions for the pattern bindings (since the new $(match_r)$ rule no longer dups the fields at runtime). Inserting dup instructions statically is essential to actually perform further dup/drop optimizations (like reuse!) on the final derived expression. Finally, we also added the DROPRU rule to reason precisely about reuse analysis described later.

Properties. Many of the properties proven for λ^1 (Reinking, Xie et al., 2021) carry over to λ^{1n}. In particular, the logical derivation rules precisely elaborate expressions with reference count operations such that they can be correctly evaluated by the target heap semantics, as stated in the following theorem:

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

We prove this theorem in separate steps: first we show soundness in a heap semantics that ignores reference count instructions (Appendix D.2), then use a separate resource calculus to show reference counts are correct (Appendix D.3), and finally combine these results for the final proof (Appendix D.4).

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 show (see Appendix D.5):

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

Note that similar to λ^1, λ^{1n} does not model *mutable references*. 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 resources in the heap, it is either reachable from the expression, or it is part of a cycle.

These theorems establish the correctness of the reference-counted heap semantics. However, correctness does not imply that evaluation is *garbage-free*. Eventually all live data is discarded but an evaluation 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 $y \mapsto^1 () \mid \mathsf{drop}\ y;\ (\lambda x.\ x)\ ()$.

5.3 Reasoning about Space with the "Star" Condition

Reinking, Xie et al. (2021) give declarative derivation rules for reference counting but then provide a separate *algorithm* that is then proven to be garbage-free. This is not ideal as it does not provide any particular insight why the algorithm is garbage-free, and if other approaches may exist as well.

By making evaluation order explicit in the normalized λ^{ln} calculus, we found a way to capture the garbage-free property *declaratively* as a single side condition on the LET rule in our new derivation rules. Moreover, by weakening the condition, this can also be used to characterize other interesting points in the design space, and provide a general framework to reason about memory consumption. In particular, we can concisely characterize frame-limited derivations. We can thus instantiate the rules by giving a specific $(\star)$ condition:

- *General* derivations $\vdash$. When we define $(\star)$ to be true, the evaluation of any derived expression is *sound* (Theorem 1).
- *Garbage-free* derivations $\vdash_{\mathsf{GF}}$. When we define the $(\star)$ condition as $\Gamma_2 \subseteq \mathsf{fv}(e_2)$, then the evaluation of any derived expression is *garbage-free* (Definition 2 and Theorem 3). At the LET rule we have the freedom to split Γ into Γ_1 and Γ_2 in any way. For garbage-free derivations though we try to minimize borrowing in the e_1 derivation (of Γ_2) and thus the condition captures intuitively that we should use the smallest Γ_2 possible, and not include variables that are not needed for the e_2 derivation.
- *Frame-limited* derivations $\vdash_{\mathsf{FL}}$. When we define $(\star)$ as $\Gamma_2 = \Gamma', \Gamma''$ where $\Gamma' \subseteq \mathsf{fv}(e_2)$ and $\mathsf{sizeof}(\Gamma'') \leqslant c$ for some constant c, then the evaluation for any derived expression is *frame-limited* (Definition 3 and Theorem 4). We define $\mathsf{sizeof}(\Gamma'')$ as the sum of the sizes of each element: $\sum_{y \in \Gamma''} \mathsf{sizeof}(y)$. This weakens the garbage-free condition to allow borrowing of any y where the runtime size of y is known to be limited by a constant. This is just enough for transformations like reuse where we borrow a reuse token until it can be reused.

Garbage-Free Evaluation. We define *garbage-free* evaluation formally as:

Definition 2. (*Garbage free evaluation*)
An evaluation $\varnothing \mid e' \longmapsto^*_r \mathsf{H} \mid x$ is called *garbage-free* iff for every intermediate state $\mathsf{H}_i \mid \mathsf{E}[v]$ in the evaluation, we have that for all $y \in \mathsf{dom}(\mathsf{H}_i)$, $\mathsf{reach}(y,\ \mathsf{H}_i \mid \lceil \mathsf{E}[v] \rceil)$.

where we use the notation $\lceil e \rceil$ to erase all drop and dup in the expression e. This is a refinement of the definition given by Reinking, Xie et al. (2021), which considered any non dup/drop steps, while we weaken this and consider only value steps v. By using $\mathsf{E}[v]$ we consider exactly those points where we are at an allocation step ($C\ \overline{x}$ or $\lambda_{\overline{z}} x.\ e$), and this is exactly the point where we want to ensure that there is no garbage. In particular, match and let are heap invariant, and applications just expand to dups, drops, and substitution. This also gives more freedom to garbage-free algorithms as it becomes possible for example to push a drop down into the branches of a match which was not possible before. Using our new definition, we can then prove (Appendix D.6):

Theorem 3. ($\vdash_{\mathsf{GF}}$ *derivations are garbage-free*)
If $\varnothing \mid \varnothing \vdash_{\mathsf{GF}} e \rightsquigarrow e'$ and $\varnothing \mid e' \longmapsto^*_r \mathsf{H} \mid x$, then the evaluation is garbage-free.

Even with the garbage-free side-condition, there are still many choice points in the derivations which gives us freedom to consider various algorithms. Generally though, when implementing Perceus one wants to dup as late as possible (push up dup into the leaves of the derivation), and do drops as early as possible. The original Perceus algorithm does this, and also trivially satisfies our garbage-free condition as it has the invariant $\Gamma \subseteq \mathsf{fv}(e)$ for any derivation step.

$\underset{\uparrow}{\mathsf{S}} \mid \underset{\uparrow}{\mathsf{R}} \Vdash \underset{\uparrow}{e} \ \rightsquigarrow \ \underset{\downarrow}{e'}$ (↑ is input, ↓ is output)

S maps variables to their heap size (if known)
R maps reuse variables r to their available heap size, $\mathsf{dom}(\mathsf{R}) \not\pitchfork \mathsf{fv}(e)$

$$\frac{}{\mathsf{S} \mid \varnothing \Vdash x \ \rightsquigarrow \ x}\ [\textsc{rvar}] \qquad \frac{\mathsf{S} \mid \mathsf{R} \Vdash e \ \rightsquigarrow \ e'}{\mathsf{S} \mid \mathsf{R}, r : n \Vdash e \ \rightsquigarrow \ \mathsf{drop}\ r;\ e'}\ [\textsc{rdropr}]$$

$$\frac{}{\mathsf{S} \mid \varnothing \Vdash C\ \overline{x} \ \rightsquigarrow \ C\ \overline{x}}\ [\textsc{rcon}] \qquad \frac{x : n\ \in \mathsf{S} \quad \mathsf{S} \mid \mathsf{R},\ r : n \Vdash e \ \rightsquigarrow \ e' \quad \text{fresh } r}{\mathsf{S} \mid \mathsf{R} \Vdash \mathsf{drop}\ x;\ e \ \rightsquigarrow \ r \leftarrow \mathsf{dropru}\ x;\ e'}\ [\textsc{rdrop-reuse}]$$

$$\frac{\mathsf{S} \mid \mathsf{R} \Vdash e \ \rightsquigarrow \ e'}{\mathsf{S} \mid \mathsf{R} \Vdash e\ x \ \rightsquigarrow \ e'\ x}\ [\textsc{rapp}] \qquad \frac{\mathsf{S} \mid \mathsf{R}_1 \Vdash e_1 \ \rightsquigarrow \ e_1' \quad \mathsf{S} \mid \mathsf{R}_2 \Vdash e_2 \ \rightsquigarrow \ e_2'}{\mathsf{S} \mid \mathsf{R}_1, \mathsf{R}_2 \Vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \ \rightsquigarrow \ \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'}\ [\textsc{rlet}]$$

$$\frac{\varnothing \mid \varnothing \Vdash e \ \rightsquigarrow \ e'}{\mathsf{S} \mid \varnothing \Vdash \lambda_\Gamma\, x.\ e \ \rightsquigarrow \ \lambda_\Gamma\, x.\ e'}\ [\textsc{rlam}] \qquad \frac{\mathsf{S} \mid \mathsf{R}, r \Vdash e \ \rightsquigarrow \ e'}{\mathsf{S} \mid \mathsf{R} \Vdash r \leftarrow \mathsf{dropru}\ x;\ e \ \rightsquigarrow \ r \leftarrow \mathsf{dropru}\ x;\ e}\ [\textsc{rdropru}]$$

$$\frac{\mathsf{S} \mid \mathsf{R} \Vdash e \ \rightsquigarrow \ e'}{\mathsf{S} \mid \mathsf{R} \Vdash \mathsf{dup}\ x;\ e \ \rightsquigarrow \ \mathsf{dup}\ x;\ e'}\ [\textsc{rdup}] \qquad \frac{\mathsf{S} \mid \mathsf{R} \Vdash e \ \rightsquigarrow \ e'}{\mathsf{S} \mid \mathsf{R} \Vdash \mathsf{drop}\ x;\ e \ \rightsquigarrow \ \mathsf{drop}\ x;\ e'}\ [\textsc{rdrop}]$$

$$\frac{\mathsf{S},\ x\ :\ n \mid \mathsf{R} \Vdash e_i \ \rightsquigarrow \ e_i' \qquad p_i\ =\ C\ x_1 \ldots x_n}{\mathsf{S} \mid \mathsf{R} \Vdash \mathsf{match}\ x\ \{\ \overline{p_i\ \mapsto e_i}\ \} \ \rightsquigarrow \ \mathsf{match}\ x\ \{\ \overline{p_i\ \mapsto e_i'}\ \}}\ [\textsc{rmatch}]$$

Fig. 7. Declarative rules for drop-guided reuse analysis

Frame-Limited Evaluation. Similar to garbage-free evaluations, we can define frame-limited evaluations:

Definition 3. (*Frame-limited evaluation*)
An evaluation $\varnothing \mid e' \longmapsto^*{}_r \mathsf{H} \mid x$ is called *frame-limited* iff for every intermediate state $\mathsf{H}_i \mid \mathsf{E}[v]$ in the evaluation, we have that H_i equals $\mathsf{H}_1, \mathsf{H}_2$ such that for all $y \in \mathsf{dom}(\mathsf{H}_1)$, $\mathsf{reach}(y, \mathsf{H}_1 \mid \lceil \mathsf{E}[v] \rceil)$ and $|\mathsf{H}_2| \leqslant c \cdot |\mathsf{E}|$ for some constant c.

This expresses that at every allocation step, we may now hold on to extra heap space H_2, but the size of H_2 is limited by a constant amount times the size of the evaluation context (i.e. the stack). We can now prove (Appendix D.7):

Theorem 4. ($\vdash_{\mathsf{FL}}$ *derivations are frame-limited*)
If $\varnothing \mid \varnothing \vdash_{\mathsf{FL}} e \ \rightsquigarrow \ e'$ and we have $\varnothing \mid e' \longmapsto^*{}_r \mathsf{H} \mid x$, then the evaluation is frame-limited with respect to the constant c chosen in the ($\star$) condition.

Borrowing a variable for a function call can also be frame-limited: Whenever an owned variable is passed as borrowed, we need to move it from the linear environment to the borrowed environment. But this can only happen in the APP and LET rules. With the APP rule, we can safely borrow since we use the variable later and with the LET rule this is frame-limited whenever the size of the borrowed variables is bounded by a constant.

5.4 Drop-Guided Reuse

Figure 7 shows the declarative derivation rules for drop-guided reuse analysis. A rule $\mathsf{S} \mid \mathsf{R} \Vdash e \ \rightsquigarrow \ e'$ states we can derive e' from e given a mapping S from variables to their heap cell size (if known), and a mapping R from reuse tokens r to their available heap size. Again, we use a multi-set for S and R; we need to ensure we only use a reuse token r once, and similar to the owned environment

Γ the leaf derivations require R to be empty (as in RVAR, RCON, and RLAM).

We state drop-guided reuse in terms of derivation rules instead of a specific algorithm in order to clearly expose the choice points. At any $\mathsf{drop}\ x;\ e$ expression we can choose to either leave it as is, or use RDROP-REUSE to try to reuse it at runtime. We can only use RDROP-REUSE though if the size of x is known statically – in our rules this only happens by matching on a particular constructor (in RMATCH) but in practice we may also use type information for example. Furthermore, in an implementation we also would only apply RDROP-REUSE if there is an actual opportunity in e for reuse to occur – that is, we look first if e contains an occurrence of $C\ x_1 .. x_n$.

Another choice point is in the RLET rule where we can freely split R into R_1 and R_2, where we may either reuse early or late if reuse is possible in both e_1 and e_2. Also, rule RDROPR is not syntax directed and thus we have a choice in what reuse token to use if there are multiple reuse tokens of the right size.

To simplify the formalization and proofs, we do not introduce a new form of constructor reuse that we used before (as `C@r`) but instead assume the runtime evaluation recognizes a pair $\mathsf{drop}\ r;\ C\ \overline{x}$ as a reuse opportunity, that is:

$\mathsf{H}, r \mapsto^1 C'\ \bot_1 \ldots \bot_n \mid \mathsf{drop}\ r;\ C\ x_1 \ldots x_n \quad \longrightarrow_r \quad \mathsf{H}, x \mapsto^1 C\ x_1 \ldots x_n \mid x$

Also, to simplify the formalization, we do not add $\bot$ either and use allocated unit constructors () instead. The rules for dropru evaluation are given in Figure 5. We can now show that drop-guided reuse is sound and frame-limited (Appendix D.8):

Theorem 5. (*Reuse is frame-limited*)
If $\Delta \mid \Gamma \vdash_{\mathsf{GF}} e \rightsquigarrow e'$, and reuse derives $\mathsf{S} \mid \mathsf{R} \Vdash e' \rightsquigarrow e''$, then $\Delta \mid \Gamma, \mathsf{R} \vdash_{\mathsf{FL}} e \rightsquigarrow e''$.

That is, if we have a garbage free derivation followed by drop-guided reuse, we could have derived that same expression directly as well using a frame-limited derivation. This is how Koka implements this as well: first a Perceus algorithm (that satisfies garbage-free derivations) followed by a drop-guided reuse algorithm (that satisfies reuse derivations).

6 BENCHMARKS

We measured the performance of drop-guided reuse in Koka, versus the previous algorithm K, and also against 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. These results are therefore mostly useful to get a sense of the absolute performance of Koka, and as evidence that our compilation techniques (Perceus, drop-guided reuse, TRMC) are viable and can be competitive.

We selected mature comparison systems that use a range of memory reclamation techniques and are considered best-in-class. We use the following functional programming languages:

- "Koka": Koka v2.3.3 with drop-guided reuse, compiling the generated C code with gcc 9.3.0 using a customized version of the mimalloc allocator (Leijen et al., 2019).
- "Koka, no trmc": As "Koka", but we disable TRMC (with "–fno-opttrmc") to measure the impact of this optimization.
- "Koka, old": As "Koka", but we use Algorithm K instead of drop-guided reuse and no borrowing (on a branch of Koka: v2.3.3-old).
- "Koka, fbip": As "Koka", but with the implementations discussed in section 7.
- Multi-core OCaml 4.12. This has a concurrent 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 (Sivaramakrishnan et al., 2020; Minsky et al., 2012, Chap.22; Doligez and Leroy, 1993).
- Haskell, GHC 8.6.5. A highly optimizing compiler with a multi generational garbage collector. We used strictness annotations in the data structures to speed up the benchmarks, as well as

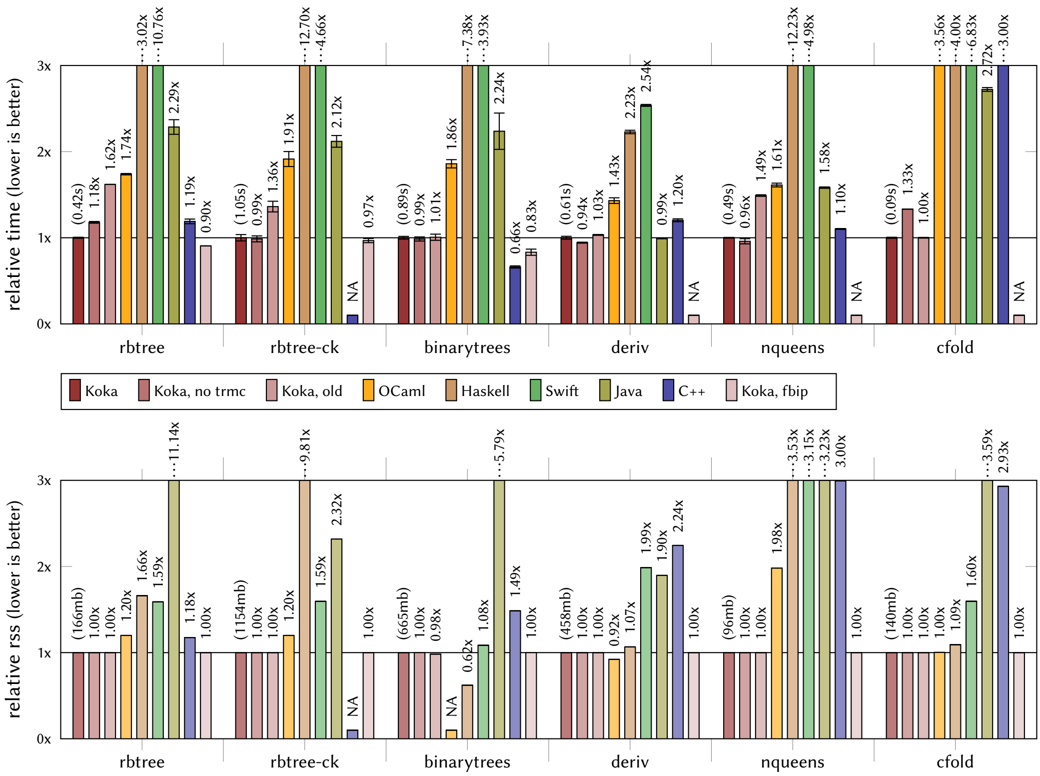

Fig. 8. All benchmarks with relative execution time and peak working set with respect to Koka. Using a 32-core x64 AMD5950X 3.4Ghz with 32GiB 3600Mhz memory (32KiB L1, 512KiB L2), Ubuntu 20.04.

to ensure that the same amount of work is done.

We also compare against Swift 5.5 and Java 17 LTS. Here, we keep the benchmarks in a functional style (without direct mutation) and only replace tail-calls with explicit loops. While Swift also uses reference counting (Ungar et al., 2017; Choi et al., 2018), Java uses the HotSpot JVM and the G1 concurrent, low-latency, generational garbage collector.

Finally, we use C++ as our performance baseline (with gcc 9.3.0 and using the standard libc allocator): For the `rbtree` benchmark we used the standard STL `std::map` implementation that uses a highly optimized in-place updating version of red-black trees (Free Software Foundation et al., 1994). The `binarytrees` benchmark use a monotonic buffer resource for memory management, while the other benchmarks (`nqueens`,`deriv`, and `cfold`) do not reclaim memory at all.

6.1 Benchmarking Balanced Trees

We use the same red-black tree benchmarks as used by Reinking, Xie et al. (2021) (which makes use of the insertion algorithm in our running example from section 2.2.1). There are two versions:

- `rbtree`: inserts 4200000 elements in tree and afterwards folds over the tree counting the `True` elements.
- `rbtree-ck`: like `rbtree` but also keeps a list of every 10th tree that is generated, effectively sharing many subtrees. This implies many subtrees have a non-unique reference count which causes many more slow paths to be taken in the Koka code. This is not done for C++ as `std::map` does not support persistence.

The results (together with other benchmarks) are shown in Figure 8 (on an AMD5950X). Koka performs surprisingly well in comparison with these mature systems, and outperforms the C++ implementation by almost 20% – how is that even possible? We conjecture this is mainly due to two factors: 1) the TRMC optimization leads to using less stack space than the C++ implementation (and perhaps less register-spills), and 2) the allocator that Koka uses can use 8-byte alignment while C++ requires 16-byte minimal alignment which leads to less allocated memory. If we compare the non-TRMC optimized version it performs only slightly better than C++ which seems to supports our thesis.

Even though this is a typical functional style algorithm both OCaml and Haskell are much slower (1.7× and 3×). The Swift version is about 10× slower even though it uses reference counting as well – this may be a combination of no reuse (due to default borrowing) and a more complex implementation of the reference counts.

The `rbtree-ck` version with many shared subtrees increases the relative speed of Koka even further. This is also somewhat surprising as it is clearly much worse for reuse (with many shared subtrees), but of course it similarly causes more pressure on garbage collectors as well as more objects get promoted to older generations. The Multi-core OCaml GC performs especially well here staying within 20% of Koka's *garbage-free* memory usage.

If we compare Koka with drop-guided reuse against Koka "old" which uses algorithm K, we can see the new algorithm does about 1.6× better on this benchmark due to the improved reuse for the `is-red` test as explained in Section 2.2.1. Finally, what is the fastest "Koka, fbip" version? We will discuss this in Section 7.

6.2 Binary Trees

The `binarytrees` benchmark comes from the Computer Language Benchmark Game (2021, The Computer Language Benchmark Game – Binarytrees Nov. 2021) which is an adaptation of Hans Boehm's `GCBench` benchmark (Boehm, 2000) (which in turn was adapted from a benchmark by John Ellis and Pete Kovac). This is an interesting benchmark as it uses *concurrent* allocation of many binary trees and calculating their checksums. Moreover, we can compare against the best performing implementations that were created by experts in each of our comparison languages.

The top 17 implementations are all languages with manual allocation (C++, C, Rust, and Free Pascal) with the top entry being C++ (#7) followed very closely by Rust and two other C++ entries (#5 and #4). For our purposes, we use C++ benchmark #5 since the performance of #7 was uneven across systems[1]. The C++ entries all use very efficient allocation by using a `monotonic_buffer_resource` for bump-pointer allocation where all nodes are freed at once at every iteration.

Figure 8 shows the benchmark results of `binarytrees` (the Koka implementation can be found in Appendix A.1). Besides C++, Koka outperforms all other languages here even though it uses a simple thread pool implementation without work-stealing. The C++ implementation is still quite a bit faster though (0.66×). Since the benchmark does concurrent allocations reference counting generally needs to be atomic. However, due to careful language design, Koka can avoid most of the overhead by checking upfront if a reference count needs to be atomic or not (Reinking, Xie et al., 2021).

6.3 Other Benchmarks

The `nqueens`, `cfold`, and `deriv` benchmarks are the same as used in the Perceus paper. We included them here for completeness but for these benchmarks the new drop guided reuse gives very similar results to algorithm K.

[1]Performance of benchmark #7 is dependent on the version of the Intel TBB library, and more than twice as slow as #5 on arm64.

The nqueens benchmark computes a list of all solutions to the N-queens problem of size 21. The large speedup here compared to Koka "old" is actually due to borrowing in the safe function. The cfold benchmark performs constant folding in program, and the deriv benchmark computes a symbolic derivative of a large expression. Again, in these benchmarks the difference with the previous reuse algorithm is minimal.

7 FBIP: FUNCTIONAL BUT IN-PLACE ALGORITHMS

Just like tail-recursion let us express loops as recursive functions, reuse analysis can be used to express imperative algorithms in a functional style. We call such algorithms *Functional But In-Place* (FBIP) (Reinking, Xie et al., 2021). In particular, it is often possible to reformulate an algorithm with non-tail-recursive calls into one that is tail recursive using an explicit visitor data type. By relying on drop-guided reuse analysis, we can now make the allocation of the visitor data type "free" by ensuring we can reuse existing objects. In this section we illustrate this technique to the rbtree and binarytrees benchmarks and show we can improve their performance even further.

7.1 Binary Trees

Most of the work in the binarytrees benchmark is in creating the trees and calculating their size using the check function:

```
type tree
  Node( left: tree, right: tree )
  Tip

fun check( t : tree ) : int
  match t
    Node(l,r) -> check(l) + check(r) + 1
    Tip       -> 0
```

This consists of two non-tail-recursive calls to check. We can improve upon this using FBIP where we use a *visitor* data type that tracks where we are in the tree. For our purposes we define:

```
type visit
  NodeR( right: tree, rest: visit )
  Done
```

We can use the new visitor datatype to write a check function that uses no extra stack space as all calls are tail-recursive:

```
fun check-fbip( t : tree, v : visit, acc : int ) : int
  match t
    Node(l,r) -> check-fbip( l, NodeR(r,v), acc + 1)  // (A)
    Tip       -> match v
      NodeR(r,v’) -> check-fbip( r, v’, acc)          // (B)
      Done        -> acc                              // (C)
```

At every Node we directly go down the left branch but remember that we still need to visit the right node by extending our visit datatype (A). When we reach a Tip we go through our visitor to now visit the saved right nodes (B) until we are done (C).

This may look more expensive, but when t happens to be unique at runtime, the NodeR allocations in (1) will reuse the Node that is matched – effectively updating these nodes in place to create a list of nodes that still need to be visited. At runtime this becomes tight loop that directly reuses the memory of the tree it is checking. Effectively, the generated code uses in-place *pointer reversal* to visit the tree without using further stack space much like stackless marking in garbage collecters with the Schorr-Waite algorithm (Schorr and Waite, 1967). In Figure 8 this is the *Koka fbip* variant and with this implementation of check Koka becomes 17% faster and within 25% of the performance of the C++ implementation (0.8× versus Koka fbip).

7.2 Balanced Trees

We can apply the same FBIP technique on our previous red-black tree balanced insertion: as we saw in Section 3.2.1, due to TRMC most calls to ins are tail recursive but not the ones that needed rebalancing. We can define a visitor for red-black trees as the *derivative* of the `tree` data type (McBride, 2001; Huet, 1997):

```
type zipper⟨a⟩
  NodeR(clr:color, l:tree⟨a⟩, key:int, value:a, zip:zipper⟨a⟩)
  NodeL(clr:color, zip:zipper⟨a⟩, key:int, value:a, r:tree⟨a⟩)
  Done
```

Using this data type we can now traverse down a tree in tail-recursive way to the insertion point, while building up the zipper that tracks where we are in the tree:

```
fun ins(t : tree⟨a⟩, k : int, v : a, z : zipper⟨a⟩) : tree⟨a⟩
  match t
    Node(c, l, kx, vx, r)
      -> if k < kx   then ins(l, k, v, NodeL(c, z, kx, vx, r))
         elif k > kx then ins(r, k, v, NodeR(c, l, kx, vx, z))
         else rebuild(z, Node(c, l, kx, vx, r))  // A
    Leaf -> balance(z, Leaf, k, v, Leaf)         // B
```

If the element is already present (A), we can use the tail-recursive `rebuild` function to reconstruct the tree using our just constructed zipper:

```
fun rebuild( z : zipper⟨a⟩, t : tree⟨a⟩ ) : tree⟨a⟩
  match z
    NodeR(c, l, k, v, z1) -> rebuild(z1, Node(c, l, k, v, t))
    NodeL(c, z1, k, v, r) -> rebuild(z1, Node(c, t, k, v, r))
    Done -> t
```

If we reach a leaf node though (B), we use `balance` to rebalance the tree going upward. Rebalancing is also tail-recursive now:

```
fun balance( z : zipper⟨a⟩, l : tree⟨a⟩, k : int, v : a, r : tree⟨a⟩ ) : tree⟨a⟩
  match z
    Done -> Node(Black,l,k,v,r)
    NodeR(Black, l1, k1, v1, z1) -> rebuild( z1, Node(Black,l1,k1,v1, Node(Red,l,k,v,r)) )
    NodeR(Red, l1, k1, v1, z1)   -> match z1
      Done -> Node(Black,l1,k1,v1, Node(Red,l,k,v,r))
      NodeR(_,l2,k2,v2,z2) -> balance( z2, Node(Black,l2,k2,v2,l1), k1, v1, Node(Black,l,k,v,r) )
      NodeL(...) -> ...
    NodeL(Black, ...) -> ...
    NodeL(Red, ...)   -> ...
```

As before, besides the inserted node, every `Node` allocation in `rebuild` and `balance` can be paired with a matched `NodeR` or `NodeL` (and the other way around in `ins`), and all can be reused in-place at runtime if the tree happens to be unique. Furthermore, for a shared tree we only allocate the zipper upfront, but the zipper itself is always unique and reused in-place by `rebuild` and `balance`.

Our novel FBIP algorithm for balanced insertion improves further upon the standard Okasaki style algorithm (Okasaki, 1999a) since we stop rebalancing as soon as we reach a `Black` node, and switch to `rebuild` instead (which is also done in the usual imperative algorithms (Guibas and Sedgewick, 1978)). The full implementation can be found in Appendix A.2. This is the *Koka fbip* variant in Figure 8 which is about 10% faster than the regular version and now around 30% faster than the C++ STL version.

8 RELATED WORK

Our work is closely based earlier work by Reinking, Xie et al. (2021) and Ullrich and de Moura (2019) (in the context of the Lean theorem prover (Moura and Ullrich, 2021)). In this work we improve upon the both of the two earlier reuse algorithms, and show that drop-guided reuse is strictly better as it is frame-limited and can find more opportunities for reuse. Our λ^{1n} calculus, heap semantics, and derivation rules differ in important details from (Reinking, Xie et al., 2021): the normalization simplifies the derivation rules and allows us to concisely express the garbage-free and frame-limited side conditions (and no longer as a particular property of a specific algorithm). It is also now immediate that the previous published Perceus algorithm is garbage-free. The new frame-limited condition allows us to reason about various transformations that are no longer garbage-free but can still be bounded.

The notion of safe-for-space was introduced by Appel (1991) and further studied by Paraskevopoulou and Appel (2019). Similar to our reuse transformation and frame-limited notion, the latter work introduces a general framework to show that the closure conversion transformation with flat environments is safe-for-space, while linked closure conversion is not. Other examples where a program transformation was proven to respect a resource bound include Crary and Weirich (2000) and Minamide (1999).

Using explicit reference count instructions in order to optimize them via static analysis is described as early as Barth (1975). Mutating unique references in place has traditionally focused on array updates (Hudak and Bloss, 1985), as in functional array languages like Sisal (McGraw et al., 1983) and SaC (Scholz, 2003; Grelck and Trojahner, 2004). Férey and Shankar (2016) provide functional array primitives that use in-place mutation if the array has a unique reference which is also present in the Koka implementation. 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 and λ^{1n} calculus are closely based on linear logic. Turner and Wadler (1999) 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. (1996) 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.

Generally, a system with linear types (Wadler, 1990), like linear Haskell (Bernardy et al., 2017), the uniqueness typing of Clean (Vries et al., 2008; Barendsen and Smetsers, 1996), or the quantitative type theory of Idris (Brady, 2021), can offer *static* guarantees that the corresponding objects are unique at runtime, so that destructive updates can always be performed safely. However, this usually also requires writing multiple versions of a function for each case (unique- versus shared argument, or an in-place mutating data structure versus a persistent one). In contrast, reuse analysis relies on dynamic runtime information, and thus reuse can be performed generally. This is also what enables FBIP to use a single function that can be used for both unique or shared objects (since the uniqueness property is *not* part of the type). These two mechanisms could be combined: if our system is extended with unique types, then reuse analysis could statically eliminate corresponding uniqueness checks.

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 (2019), 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. Ungar et al. (2017) optimize atomic reference counts by tagging objects that can be potentially thread-shared. Later work by Choi et al. (2018), uses *biased* reference counting to avoid many atomic updates. They remove the need for tagging by extending the object header with the ID of the owning thread together with two reference counts: a shared one that needs atomic updates, and the biased one for the owning thread.

The CPython implementation also uses reference counting, and uses ownership-based reference counts for parameters but still only drops the reference count of local variables when exiting the frame. 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 (Yarantsev, 2020). 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 (1976) 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 (2006) 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.

9 CONCLUSION AND FUTURE WORK

In this work we show the effectiveness of drop-guided reuse, and give a precise characterization of garbage-free and frame-limited evaluations. However, this works well partly because in a functional style language like Koka it is uncommon to create cycles (which can only be created through mutable references). We would like to see if we can combine some of the static analysis with cycle collection. Also, we are interested in language support to guarantee that reuse is happening at compile time.

REFERENCES

Andrew W. Appel. 1991. *Compiling with Continuations*. Cambridge University Press. doi:10.1017/CBO9780511609619.

Erik Barendsen, and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6 (6). Cambridge University Press: 579–612. doi:10.1017/S0960129500070109.

Jeffrey M. Barth. Jun. 1975. *Shifting Garbage Collection Overhead to Compile Time*. UCB/ERL M524. EECS Department, University of California, Berkeley. `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1975/29109.html`.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Dec. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2 (POPL). doi:10.1145/3158093.

Hans Boehm. 2000. GC Bench. `https://hboehm.info/gc/gc_bench`.

Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, edited by Anders Møller and Manu Sridharan, 194:9:1–9:26. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany. doi:10.4230/LIPIcs.ECOOP.2021.9.

Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. 1996. Reference Counting as a Computational Interpretation of Linear Logic. *Journal of Functional Programming* 6: 6–2.

Jiho Choi, Thomas Shull, and Josep Torrellas. 2018. Biased Reference Counting: Minimizing Atomic Operations in Garbage Collection. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus. doi:10.1145/3243176.3243195.

George E Collins. 1960. A Method for Overlapping and Erasure of Lists. *Communications of the ACM* 3 (12). ACM New York, NY, USA: 655–657.

Karl Crary, and Stephnie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 184–198. POPL '00. Boston, MA, USA. doi:10.1145/325694.325716.

L. Peter Deutsch, and Daniel G. Bobrow. Sep. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM* 19 (9): 522–526. doi:10.1145/360336.360345.

Damien Doligez, and Xavier Leroy. Jan. 1993. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL)*, 113–123. ACM press.

Gaspard Férey, and Natarajan Shankar. 2016. Code Generation Using a Formal Model of Reference Counting. In *NASA Formal Methods*, edited by Sanjai Rayadurgam and Oksana Tkachuk, 150–165. Springer International Publishing.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 237–247. PLDI '93. Albuquerque, New Mexico, USA. doi:10.1145/155090.155113.

Free Software Foundation, Silicon Graphics, and Hewlett–Packard Company. 1994. Internal Red-Black Tree Implemention for "Stl::map." `https://github.com/gcc-mirror/gcc/tree/master/libstdc++-v3/src/c++98/tree.cc`.

Matt Gallagher. Dec. 2016. Reference Counted Releases in Swift. `https://www.cocoawithlove.com/blog/resources-releases-reentrancy.html`. Blog post.

Clemens Grelck, and Kai Trojahner. Sep. 2004. Implicit Memory Management for SAC. In *6th International Workshop on Implementation and Application of Functional Languages (IFL'04)*. Lübeck, Germany.

Leo J Guibas, and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, 8–21. IEEE.

Paul Hudak, and Adrienne Bloss. 1985. The Aggregate Update Problem in Functional Programming Systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 300–314. POPL '85. ACM, New Orleans, Louisiana, USA. doi:10.1145/318593.318660.

Gérard P. Huet. 1997. The Zipper. *Journal of Functional Programming* 7 (5): 549–554.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. doi:10.4204/EPTCS.153.8.

Daan Leijen. Jan. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. doi:10.1145/3009837.3009872.

Daan Leijen. 2021. The Koka Language. `https://koka-lang.github.io`.

Daan Leijen, Zorn Ben, and Leo de Moura. 2019. Mimalloc: Free List Sharding in Action. *Programming Languages and Systems*, LNCS, 11893. Springer International Publishing. doi:10.1007/978-3-030-34175-6_13. APLAS'19.

Yossi Levanoni, and Erez Petrank. Jan. 2006. An On-the-Fly Reference-Counting Garbage Collector for Java. *ACM Trans. Program. Lang. Syst.* 28 (1): 1–69. doi:10.1145/1111596.1111597.

Conor McBride. 2001. The Derivative of a Regular Type Is Its Type of One-Hole Contexts. `http://strictlypositive.org/diff.pdf`. (Extended Abstract).

J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. Jul. 1983. *SISAL: Streams and Iteration in a Single-Assignment Language. Language Reference Manual, Version 1. 1*. LLL/M-146, ON: DE83016576. Lawrence Livermore National Lab., CA, USA.

Yasuhiko Minamide. 1999. Space-Profiling Semantics of the Call-by-Value Lambda Calculus and the CPS Transformation. *Electronic Notes in Theoretical Computer Science* 26: 105–120. doi:10.1016/S1571-0661(05)80286-5. HOOTS '99, Higher Order Operational Techniques in Semantics.

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2012. *Real World OCaml: Functional Programming for the Masses*. `https://dev.realworldocaml.org`.

Leonardo de Moura, and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, edited by André Platzer and Geoff Sutcliffe, 625–635.

Chris Okasaki. 1999. Red-Black Trees in a Functional Setting. *Journal of Functional Programming* 9 (4). Cambridge University Press: 471–477. doi:10.1017/S0956796899003494.

Chris Okasaki. Jun. 1999. *Purely Functional Data Structures*. Colombia University, New York.

Zoe Paraskevopoulou, and Andrew W Appel. 2019. Closure Conversion Is Safe for Space. *Proceedings of the ACM on Programming Languages* 3 (ICFP). ACM New York, NY, USA: 1–29.

Gordon D. Plotkin, and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11 (1): 69–94. doi:10.1023/A:1023064908962.

Gordon D. Plotkin, and Matija Pretnar. Mar. 2009. Handlers of Algebraic Effects. In *18th European Symposium on Programming Languages and Systems*, 80–94. ESOP'09. York, UK. doi:10.1007/978-3-642-00590-9_7.

Reinking, Xie, de Moura, and Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 96–111. PLDI 2021. ACK, New York, NY, USA. doi:10.1145/3453483.3454032.

Sven-Bodo Scholz. Nov. 2003. Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming* 13 (6). Cambridge University Press, USA: 1005–1059. doi:10.1017/S0956796802004458.

H. Schorr, and W. M. Waite. Aug. 1967. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Comm. ACM* 10 (8). ACM: 501–506. doi:10.1145/363534.363554.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Aug. 2020. Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.* 4 (ICFP). ACM. doi:10.1145/3408995.

The Computer Language Benchmark Game – Binarytrees. Nov. 2021. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html`.

David N. Turner, and Phillip Wadler. 1999. Operational Interpretations of Linear Logic, number 227: 231–248.

Sebastian Ullrich, and Leonardo de Moura. Sep. 2019. 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)*. Singapore.

David Ungar, David Grove, and Hubertus Franke. 2017. Dynamic Atomicity: Optimizing Swift Memory Management. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, 15–26. DLS 2017. Vancouver, BC, Canada. doi:10.1145/3133841.3133843.

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2008. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages (IFL'08)*, edited by Olaf Chitil, Zoltán Horváth, and Viktória Zsók, 201–218. Springer.

Phillip Wadler. 1990. Linear Types Can Change the World! In *Programming Concepts and Methods*.

Andrew K. Wright, and Matthias Felleisen. Nov. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115 (1): 38–94. doi:10.1006/inco.1994.1093.

Ningning Xie, and Daan Leijen. Aug. 2021. Generized Evidence Passing for Effect Handlers – Efficient Compilation of Effect Handlers to C. In *Proceedings of the 26th ACM SIGPLAN International Conference on Functional Programming (ICFP'2021)*. ICFP '21. Virtual.

Danil Yarantsev. Oct. 2020. ORC - Nim's Cycle Collector. `https://nim-lang.org/blog/2020/10/15/introduction-to-arc-orc-in-nim.html`.

APPENDIX

A SOURCES

A.1 Binary Trees in Koka

This is the implementation of the `binarytrees` benchmark of the Computer Language Benchmark Game (2021, The Computer Language Benchmark Game – Binarytrees Nov. 2021):

```
import std/os/env
import std/os/task

type tree
  Node( left : tree, right : tree )
  Tip

fun make( depth : int ) : div tree
  if depth >= 1
    then Node( make(depth - 1), make(depth - 1) )
    else Node( Tip, Tip )

fun check( t : tree ) : int
  match t
    Node(l,r) -> l.check + r.check + 1
    Tip       -> 0

fun sum-count( count : int, depth : int ) : div int
  fold-int(count+1,0) fn(i,csum)
    csum + make(depth).check

fun gen-depth( min-depth : int, max-depth : int ) : pure list⟨(int,int,promise⟨int⟩)⟩
  list(min-depth, max-depth, 2) fn(d)          // list from min to max with stride 2
    val count = 2^(max-depth + min-depth - d)
    (count, d, task{ sum-count(count, d) } )   // one task per depth

fun show( msg : string, depth : int, check : int ) : console ()
  println(msg ++ " of depth " ++ depth.show ++ "\t check: " ++ check.show)

public fun main()
  val n = get-args().head.default("").parse-int.default(21)
  val min-depth = 4
  val max-depth = max(min-depth + 2, n)

  // allocate and free the stretch tree
  val stretch-depth = max-depth.inc
  show( "stretch tree", stretch-depth, make(stretch-depth).check )

  // allocate long lived tree
  val long = make(max-depth)

  // allocate and free many trees in parallel
  val trees = gen-depth( min-depth, max-depth )
  trees.foreach fn((count,depth,csum))
    show( count.show ++ "\t trees", depth, csum.await )

  // and check if the long lived tree is still good
  show( "long lived tree", max-depth, long.check )
```

A.2 Red-Black Trees with FBIP

This is the implementation of red-black tree rebalancing using the FBIP approach described in Section 7. Note how `move-up`, `balance-red`, `ins` are all nicely tail-recursive:

```
type color
  Red
  Black

type tree⟨a⟩
  Node(color : color, lchild : tree⟨a⟩, key : int, value : a, rchild : tree⟨a⟩)
  Leaf
```

```
type zipper⟨a⟩
  NodeR(color : color, lchild : tree⟨a⟩, key : int, value : a, zip : zipper⟨a⟩)
  NodeL(color : color, zip : zipper⟨a⟩, key : int, value : a, rchild : tree⟨a⟩)
  Done

fun rebuild(z : zipper⟨a⟩, t : tree⟨a⟩) : tree⟨a⟩
  match z
    NodeR(c, l, k, v, z1) -> rebuild(z1, Node(c, l, k, v, t))
    NodeL(c, z1, k, v, r) -> rebuild(z1, Node(c, t, k, v, r))
    Done -> t

fun balance( z : zipper⟨a⟩, l : tree⟨a⟩, k : int, v : a, r : tree⟨a⟩ ) : tree⟨a⟩
  match z
    NodeR(Black, l1, k1, v1, z1) -> rebuild( z1, Node( Black, l1, k1, v1, Node(Red,l,k,v,r) ) )
    NodeL(Black, z1, k1, v1, r1) -> rebuild( z1, Node( Black, Node(Red,l,k,v,r), k1, v1, r1 ) )
    NodeR(Red, l1, k1, v1, z1) -> match z1
      NodeR(_,l2,k2,v2,z2) -> balance( z2, Node(Black,l2,k2,v2,l1), k1, v1, Node(Black,l,k,v,r) )
      NodeL(_,z2,k2,v2,r2) -> balance( z2, Node(Black,l1,k1,v1,l), k, v, Node(Black,r,k2,v2,r2) )
      Done -> Node(Black, l1, k1, v1, Node(Red,l,k,v,r))
    NodeL(Red, z1, k1, v1, r1) -> match z1
      NodeR(_,l2,k2,v2,z2) -> balance( z2, Node(Black,l2,k2,v2,l), k, v, Node(Black,r,k1,v1,r1) )
      NodeL(_,z2,k2,v2,r2) -> balance( z2, Node(Black,l,k,v,r), k1, v1, Node(Black,r1,k2,v2,r2) )
      Done -> Node(Black, Node(Red,l,k,v,r), k1, v1, r1)
    Done -> Node(Black,l,k,v,r)

fun ins(t : tree⟨a⟩, k : int, v : a, z : zipper⟨a⟩) : tree⟨a⟩
  match t
    Node(c, l, kx, vx, r)
      -> if k < kx   then ins(l, k, v, NodeL(c, z, kx, vx, r))
         elif k > kx then ins(r, k, v, NodeR(c, l, kx, vx, z))
         else rebuild(z, Node(c, l, kx, vx, r))
    Leaf -> balance(z, Leaf, k, v, Leaf)

fun insert(t : tree⟨a⟩, k : int, v : a) : tree⟨a⟩
  ins(t, k, v, Done)
```

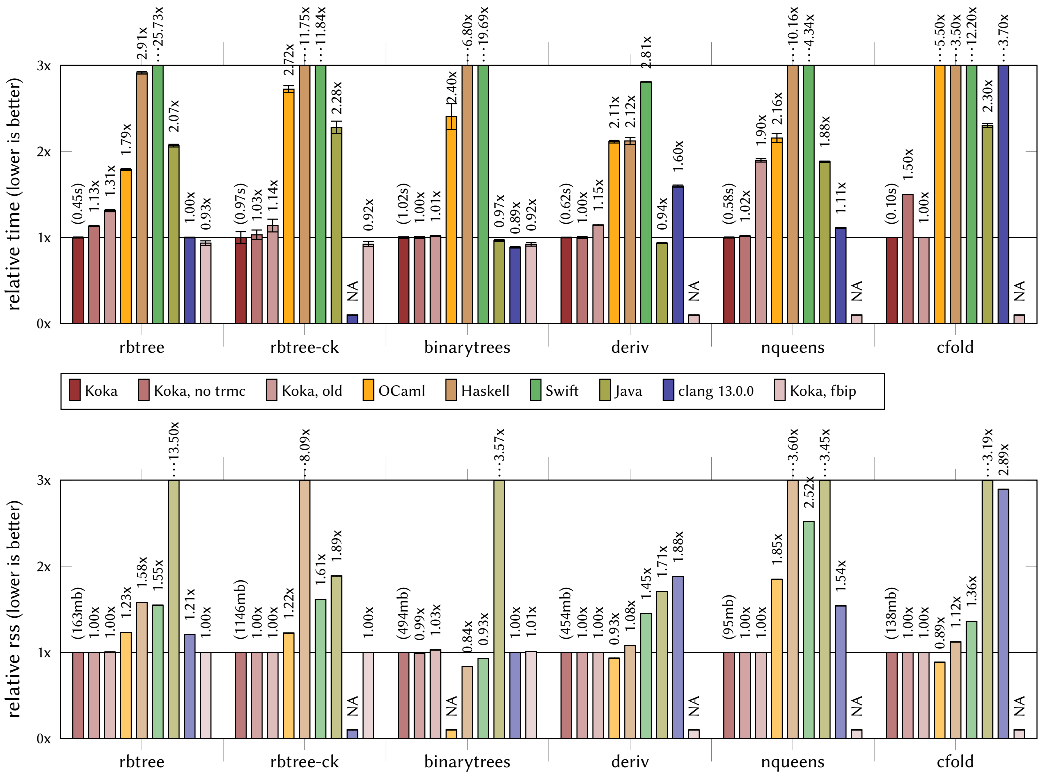

Fig. 9. Relative execution time and peak working set with respect to Koka. Using an 8-core arm64 M1 3.2Ghz (192+128KiB L1, 12MiB shared L2) with 16GiB 4266Mhz memory, macOS 11.6

B BENCHMARKS ON AN ARM64 M1

Figure 9 gives the benchmark results on an arm64 M1. The relative execution times are roughly similar to that of the AMD5950X except for the `rbtree` and `binarytrees` benchmarks. In the `rbtree` benchmark, Koka now has similar performance as C++. This may be partly the use of the clang (vs gcc) which uses a slightly different implementation of red-black tree rebalancing in `std::map` [2]. Another factor may be the the larger shared L2 cache on the M1. Also, TRMC seems relatively less effective which may be due to slightly faster call frames on the M1.

In the `binarytrees` benchmark, Koka is now just 11% slower as C++ #5 and the fbip version is almost as fast. This may be mostly due to the concurrency being limited to 8 cores. All benchmark implementations other than the C++ one are limited in concurrency to about ~10× due to the *depth* steps in the benchmark. The C++ implementation also uses parallelism within the iterations per depth-step so on the 32-core AMD5950X this may account partly for the improved performance.

C EXAMPLES OF WORST-CASE HEAP USAGE

C.1 Borrowing is not frame-limited

In general, borrowing is not frame-limited. For example, the function `go` below does not use its argument `xs`, but marks it as borrowed so it needs to stay live. It then calls itself recursively with

[2]The LLVM `std::map` implementation can be found at https://github.com/llvm/llvm-project/blob/main/libcxx/include/__tree

another `big-list()` for `xs`.

```
fun big-list()
  list(1, 100)

fun go( n, ^xs )
  val y = big-list()
  if(n >= 1)
    then go(n - 1, y)
    else []

fun main(n)
  go(n, big-list())
```

If we would ignore the borrow annotation on `xs`, we could evaluate this program with only one `big-list()` live at a time. With the borrow annotation, the program keeps up to `n` many `big-list()`s alive at a time. In a certain sense this program illustrates a worst-case: Any value we borrow must have been constructed before and so the heap usage can not increase due to a single borrowing function call by more than the maximum heap usage of the non-borrowing program. In other words, if H_{max} denotes the maximum heap size of the non-borrowing program, the heap usage of a program with borrowing is bounded at any evaluation step by H_{max} times the number of stack frames.

C.2 Algorithm D is not frame-limited

We can modify the program from the last section to show that the reuse Algorithm D (Ullrich and de Moura, 2019) can use the same amount of memory as borrowing:

```
fun big-list()
  list(1, 100)

fun go( n, xs )
  val y = big-list()
  if(n >= 1)
    then match y
          Cons(_,_) ->
            go(n - 1, y)
            // dropru(y) is inserted here
            Cons(1, [])
          Nil -> [] // won't happen
    else []

fun main(n)
  go(n, big-list())
```

Unlike previously, `xs` is not borrowed and so the first `big-list()` generated in `main` will be discarded. But in the subsequent iterations, `y` will be held on to at the callsite so that it can be reused for the `Cons(1, [])` allocation.

C.3 Lean's implementation of Algorithm D is not frame-limited

In practice, Lean (Moura and Ullrich, 2021) performs several transformations that make it difficult to see that the problematic program outlined above for Algorithm D also applies to their implementation. In particular:

- Lean lifts fully applied constructors like `Cons(1, [])` to a definition, so that they only have to be allocated once. We therefore ask the user for input and compute the tail `[]` based on this input. This ensures that no inlining can lead the constructor to be fully applied.
- An unused argument like `xs` for `go` is always be marked as borrowed. We print it to the console so that it is marked as owned.
- A pure function call of which we don't use the result like `go(n - 1, y)` above would be eliminated. Thanks to the modifications of the last two bullets, the function is not pure and will not get eliminated.

- The result of `big_list` is not recomputed every time we call it if we only pass `()`. We take the length of the user-provided string to compute the length of `big_list` so that the recomputation has to happen.

Keeping the above in mind, we can write the example as:

```
open List

set_option trace.compiler.ir.result true

def big_list (n : Nat) : List Nat :=
  List.range n

def print_head (xs : List Nat) :=
  match xs with
    | (cons h _) ⇒ IO.println (toString h)
    | nil ⇒ IO.println "Nil"

def get_tail : IO (List Nat × Nat) := do
  let i <- IO.getStdin
  let s <- i.getLine
  match String.toList s with
    | (cons _ _) ⇒ (cons 1 [], String.length s)
    | nil ⇒ (nil, 1)

partial def go (n : Nat) (xs : List Nat) : IO (List Nat) := do
  print_head xs
  let (t, n) <- get_tail
  let y := big_list n;
  if n >= 1 then
    match y with
      | (cons _ _) ⇒ do
          let _ <- go (n - 1) y
          return cons 1 t
      | nil ⇒ return t
  else return t

def main : IO Unit := do
  let _ <- go 10 (big_list 10)
  IO.println "Done"
```

As of this writing (Lean version 4.0.0, commit 6475e3d5ccaf, Release) Lean emits the following IR:

```
...
let x_12 : obj := List.range x_11; // inlined big_list
...
inc x_12; // hold on to the list
let x_16 : obj := go x_15 x_12 x_9; // recursive call with (n - 1), the list and the IO state
...
let x_18 : obj := proj[1] x_12; // drop the children of the first list cell
dec x_18;
let x_19 : obj := proj[0] x_12;
dec x_19;
...
set x_12[1] := x_10; // use the list cell for the 'cons 1 t' constructor
set x_12[0] := x_13;
...
```

$$
\begin{array}{l}
\mathsf{H}: \quad x \mapsto (\mathbb{N}^{+}, v) \\
\mathsf{E} ::= \Box \;\mid \mathsf{E}\; e \;\mid x\; \mathsf{E} \mid C\; x_1 \ldots x_{i-1}\; \mathsf{E}\; v_{i+1} \ldots v_n \\
\quad \mid \quad \mathsf{let}\; x \;=\; \mathsf{E}\; \mathsf{in}\; e \mid \mathsf{match}\; \mathsf{E}\; \{\; \overline{p_i \to e_i}\; \}
\end{array}
\qquad
\frac{\mathsf{H} \mid e \longrightarrow_h \mathsf{H}' \mid e'}{\mathsf{H} \mid \mathsf{E}[e] \longmapsto_h \mathsf{H}' \mid \mathsf{E}[e']}\;[\textsc{eval}]
$$

Evaluation steps:

$$
\begin{array}{llcll}
(\mathit{lam}_h) & \mathsf{H} \mid \lambda_{\overline{z}}\; x.\; e & \longrightarrow_h & \mathsf{H}, f \mapsto^1 \lambda_{\overline{z}}\; x.\; e \quad \mid f & \mathsf{fresh}\; f \\
(\mathit{con}_h) & \mathsf{H} \mid C\; x_1 \ldots x_n & \longrightarrow_h & \mathsf{H}, z \mapsto^1 C\; x_1 \ldots x_n \;\mid z & \mathsf{fresh}\; z \\
(\mathit{app}_h) & \mathsf{H} \mid f\; y & \longrightarrow_h & \mathsf{H} \mid \mathsf{dup}\; \overline{z};\; \mathsf{drop}\; x;\; e[x{:=}y] & (f \mapsto^n \lambda_{\overline{z}}\; x.\; e) \in \mathsf{H} \\
(\mathit{match}_h) & \mathsf{H} \mid \mathsf{match}\; y\; \{\overline{p_i \to e_i}\} & \longrightarrow_h & \mathsf{H} \mid e_i[\overline{x}{:=}\overline{z}] & \text{with } p_i \;=\; C\; \overline{x} \text{ and } (y \mapsto^n C \\
(\mathit{let}_h) & \mathsf{H} \mid \mathsf{let}\; x \;=\; z\; \mathsf{in}\; e & \longrightarrow_h & \mathsf{H} \mid e[x{:=}z] & \\
(\mathit{dup}_h) & \mathsf{H} \mid \mathsf{dup}\; x;\; e & \longrightarrow_h & \mathsf{H} \mid e & \\
(\mathit{drop}_h) & \mathsf{H} \mid \mathsf{drop}\; x;\; e & \longrightarrow_h & \mathsf{H} \mid e & \\
(\mathit{dropru}_h) & \mathsf{H} \mid r \leftarrow \mathsf{dropru}\; x;\; e & \longrightarrow_h & \mathsf{H}, z \mapsto^1 () \mid e[r{:=}z] & \mathsf{fresh}\; z
\end{array}
$$

Fig. 10. Non-reference-counted heap semantics for λ^{1n}.

D PROOFS

D.1 Extending strict evaluation semantics

If we add $\mathsf{dup}\; e$ and $\mathsf{drop}\; e$ in the syntax, as well as add to the standard semantics in Figure 2 the following rules:

$$
\begin{array}{llcl}
(\mathit{dup}) & \mathsf{dup}\; e';\; e & \longrightarrow & e \\
(\mathit{drop}) & \mathsf{drop}\; e';\; e & \longrightarrow & e \\
(\mathit{dropru}) & r \leftarrow \mathsf{dropru}\; e';\; e & \longrightarrow & e
\end{array}
$$

we immediately see that translations do not change evaluation:

Theorem 6. (*Translation is sound*)
If $e \longmapsto^{*} v$ with $\varnothing \mid \varnothing \vdash e \;\rightsquigarrow\; e'$, then also $e' \longmapsto^{*} v$.

Proof. (*Of Theorem 6*) Follows directly from Lemma 1 and the three reduction rules (dup), (drop) and (dropru). Note that the expression e can not depend on a reuse token r, since they are fresh. Therefore r will only appear as the result of a dropru and as the argument to a dup, drop or dropru. □

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

Proof. (*Of Lemma 1*) By straightforward case analysis of each derivation. □

D.2 Soundness of non-reference-counted heap semantics

To show that our logical rules are sound and leave no garbage, we extend the formalization of (Reinking, Xie et al., 2021). Lemma 9 and 10 of their proof rely on the heap evaluation context being not too different from the standard evaluation context. Unfortunately, this is not the case in our formalization: Our heap evaluation context only works for normalized terms and evaluation may make normalized terms only partially normalized. We also can not just extend the evaluation context for the heap semantics, because the match rule relies on the scrutinee being a variable. We will therefore first show that a heap semantics that ignores dups, drops and drop-reuses (Figure 10) will evaluate an expression to the same value as the standard strict evaluation.

Lemma 2.
Let $\mathsf{H} \mid e \longmapsto_h \mathsf{H}' \mid e'$ be a heap derivation and z a fresh variable. Then $\mathsf{H}, z \mapsto^n v \mid e \longmapsto_h \mathsf{H}', z \mapsto^n v \mid e'$.

Proof. By straightforward case analysis on the evaluation rules.

Lemma 3.
Let H be a heap and v a value. Then $\mathsf{H} \mid v \longmapsto^*_h \mathsf{H}' \mid x$ with $[\mathsf{H}']x \;=\; [\mathsf{H}]v$.

Proof. By induction on v.

case x.

$\mathsf{H} \mid x \longmapsto^*_h \mathsf{H} \mid x$ (1), given
$[\mathsf{H}]x \;=\; [\mathsf{H}]x$ (2), trivial

case $[\lambda x.\ e]$.

$\mathsf{H} \mid \lambda x.\ e \longmapsto^*_h \mathsf{H}, f \mapsto^1 \lambda x.\ e \mid f$ (1), by (lam_h)
$[\mathsf{H}, f \mapsto^1 \lambda x.\ e]f \;=\; [\mathsf{H}](\lambda x.\ e)$ (2), by (1)

case $[C\ \overline{v}]$.

$\mathsf{H} \mid C\ \overline{v} \longmapsto^*_h \mathsf{H}, \overline{x} \mapsto^1 \overline{v} \mid C\ \overline{x}$ (1), by the evaluation context
$[\mathsf{H}]\overline{v} \;=\; [\mathsf{H}, \overline{x} \mapsto^1 \overline{v}]\overline{x}$ (2), by induction
$\mathsf{H}, \overline{x} \mapsto^1 \overline{v} \mid C\ \overline{x} \longmapsto^*_h \mathsf{H}, \overline{x} \mapsto^1 \overline{v}, x \mapsto^1 C\ \overline{x} \mid x$ (3), by (con_h)
$[\mathsf{H}, \overline{x} \mapsto^1 \overline{v}, x \mapsto^1 C\ \overline{x}]x \;=\; [\mathsf{H}](C\ \overline{v})$ (4), by (2)

Lemma 4.
Let H be a heap and e an expression. If $[\mathsf{H}]e \longmapsto^* v$, then $\mathsf{H} \mid e \longmapsto^*_h \mathsf{H}' \mid x$ with $[\mathsf{H}']x \;=\; v$.

Proof. By induction on the judgment $[\mathsf{H}]e \longmapsto^* v$, where $\longmapsto^*$ is the reflexive transitive closure of $\longmapsto$.

case $[\ [\mathsf{H}]v \longmapsto^* [\mathsf{H}]v]$ (reflexive case).

$[\mathsf{H}]v \longmapsto^* [\mathsf{H}]v$ (1), given
$\mathsf{H} \mid v \longmapsto^*_h \mathsf{H}' \mid x$ (2), by lemma 3
$[\mathsf{H}']x \;=\; [\mathsf{H}]v$ (3), by lemma 3

case $[\ [\mathsf{H}]e \longmapsto [\mathsf{H}]e'$ and $[\mathsf{H}]e' \longmapsto^* v]$ (transitive case).

We proceed with induction on the judgment $[\mathsf{H}]e \longmapsto [\mathsf{H}]e'$.

case $[\mathsf{E} \;=\; \Box$ and $e \;=\; (\lambda x.\ \underline{e})\ v]$.

$e' \;=\; \underline{e}[x{:=}v]$ (1), given
$\mathsf{H} \mid (\lambda x.\ \underline{e})\ v \longmapsto_h \mathsf{H}, f \mapsto^1 \lambda x.\ \underline{e} \mid f\ v$ (2), by (lam_h)
$\mathsf{H}, f \mapsto^1 \lambda x.\ \underline{e} \mid f\ v \longmapsto_h \mathsf{H}', f \mapsto^1 \lambda x.\ \underline{e} \mid f\ z$ (3), by lemma 3
$\mathsf{H}', f \mapsto^1 \lambda x.\ \underline{e} \mid f\ z \longmapsto^*_h \mathsf{H}', f \mapsto^1 \lambda x.\ \underline{e} \mid \underline{e}[x{:=}z]$ (4), by (app_h), (dup_h), $(drop_h)$
$[\mathsf{H}]v \;=\; [\mathsf{H}']z$ (5), by lemma 3
$[\mathsf{H}]e' \;=\; [\mathsf{H}, z \mapsto^1 v]\underline{e}[x{:=}z] \;=\; [\mathsf{H}']\underline{e}[x{:=}z]$ (6), by (1) and (5)
$[\mathsf{H}]e' \;=\; [\mathsf{H}', f \mapsto^1 \lambda x.\ \underline{e}]\underline{e}[x{:=}z]$ (7), by monotonicity

case $[\mathsf{E} \;=\; \Box$ and $e \;=\; \mathsf{let}\ x \;=\; v\ \mathsf{in}\ \underline{e}]$.

$e' \;=\; \underline{e}[x{:=}v]$ (1), given
$\mathsf{H} \mid \mathsf{let}\ x \;=\; v\ \mathsf{in}\ \underline{e} \longmapsto_h \mathsf{H}' \mid \mathsf{let}\ x \;=\; z\ \mathsf{in}\ \underline{e}$ (2), by lemma 3
$\mathsf{H}' \mid \mathsf{let}\ x \;=\; z\ \mathsf{in}\ \underline{e} \longmapsto_h \mathsf{H}' \mid \underline{e}[x{:=}z]$ (3), by (let_h)
$[\mathsf{H}]v \;=\; [\mathsf{H}']z$ (4), by lemma 3
$[\mathsf{H}]e' \;=\; [\mathsf{H}, z \mapsto^1 v]\underline{e}[x{:=}z] \;=\; [\mathsf{H}']\underline{e}[x{:=}z]$ (5), by (1) and (4)

case $[\mathsf{E} \;=\; \Box$ and $e \;=\; \mathsf{match}\ (C\ \overline{v})\ \{\overline{p_i \to e_i}\}]$.

$e' = e_i[\overline{x}{:=}\overline{v}]$	(1), given where $p_i = C\ \overline{x}$
$\mathsf{H} \mid \mathsf{match}\ (C\ \overline{v})\ \{\overline{p_i \to e_i}\} \longmapsto_h \mathsf{H}' \mid \mathsf{match}\ z\ \{\overline{p_i \to e_i}\}$	(2), by lemma 3
$[\mathsf{H}](C\ \overline{v}) = [\mathsf{H}']z$	(3), by lemma 3
$(z \mapsto^n C\ \overline{z}) \in \mathsf{H}'$	(4), by (3)
$\mathsf{H}' \mid \mathsf{match}\ z\ \{\overline{p_i \to e_i}\} \longmapsto_h \mathsf{H}' \mid e_i[\overline{x}{:=}\overline{z}]$	(5), by $(match_h)$ and (4)
$[\mathsf{H}]e' = [\mathsf{H}, \overline{z} \mapsto^1 \overline{v}]e_i[\overline{x}{:=}\overline{z}] = [\mathsf{H}']e_i[\overline{x}{:=}\overline{z}]$	(6), by (1) and (3)

case $[\mathsf{E} = \Box$ and $e = \mathsf{dup}\ x;\ e_2]$.

$e' = e_2$	(1), given
$[\mathsf{H}]e' \longmapsto^* v$	(2), given
$\mathsf{H} \mid e' \longmapsto^* \mathsf{H}' \mid x$	(3), by induction
$[\mathsf{H}']x = v$	(4), by induction

case $[\mathsf{E} = \Box$ and $e = \mathsf{drop}\ x;\ e_2]$.

$e' = e_2$	(1), given
$[\mathsf{H}]e' \longmapsto^* v$	(2), given
$\mathsf{H} \mid e' \longmapsto^* \mathsf{H}' \mid x$	(3), by induction
$[\mathsf{H}']x = v$	(4), by induction

case $[\mathsf{E} = \Box$ and $e = r \leftarrow \mathsf{dropru}\ x;\ e_2]$.

$e' = e_2$	(1), given
$[\mathsf{H}]e' \longmapsto^* v$	(2), given
$\mathsf{H} \mid e' \longmapsto^* \mathsf{H}' \mid x$	(3), by induction
$[\mathsf{H}']x = v$	(4), by induction
$\mathsf{H}, z \mapsto^1 () \mid e' \longmapsto^* \mathsf{H}', z \mapsto^1 () \mid x$	(5), by (3) as z fresh
$[\mathsf{H}', z \mapsto^1 ()]x = v$	(6), by (4) and lemma 2

case $[\mathsf{E} = \mathsf{E}_1\ e''$ and $e = \underline{e}\ e'']$.

$e' = w\ e''$	(1), given
$[\mathsf{H}]\underline{e} \longmapsto^* w$	(2), given
$\mathsf{H} \mid \underline{e} \longmapsto^*{}_h \mathsf{H}' \mid x$	(3), by induction
$[\mathsf{H}]w = [\mathsf{H}']x$	(4), by induction
$[\mathsf{H}]e' = [\mathsf{H}'](x\ e'')$	(5), by (4)
$[\mathsf{H}]e' \longmapsto^* v$	(6), given
$[\mathsf{H}'](x\ e'') \longmapsto^* v$	(7), by (5)
$\mathsf{H}' \mid (x\ e'') \longmapsto^*{}_h \mathsf{H}'' \mid y$	(8), by induction
$[\mathsf{H}]v = [\mathsf{H}'']y$	(9), by induction
$\mathsf{H} \mid (\underline{e}\ e'') \longmapsto^*{}_h \mathsf{H}'' \mid y$	(10), by (3) and (8)
$\mathsf{H} \mid e \longmapsto^*{}_h \mathsf{H}'' \mid y$	(11), by (10)

case $[\mathsf{E} = v\ \mathsf{E}_1$ and $e = v\ e'']$.

$e' = v\ w$ (1), given
$\mathsf{H} \mid v \longmapsto^*{}_h \mathsf{H}' \mid x$ (2), by lemma 3
$[\mathsf{H}']x = [\mathsf{H}]v$ (3), by lemma 3
$[\mathsf{H}']w = [\mathsf{H}]w$ (4), since new names are fresh
$\mathsf{H}' \mid e'' \longmapsto^*{}_h \mathsf{H}'' \mid y$ (5), by induction
$[\mathsf{H}'']y = [\mathsf{H}]w$ (6), by induction and (4)
$[\mathsf{H}](v\ w) \longmapsto^* u$ (7), given
$[\mathsf{H}''](x\ y) \longmapsto^* u$ (8), by (7),(3),(6)
$\mathsf{H}'' \mid x\ y \longmapsto^*{}_h \mathsf{H}_3 \mid z$ (9), by induction
$[\mathsf{H}_3]z = u$ (10), by induction
$\mathsf{H} \mid v\ e'' \longmapsto^*{}_h \mathsf{H}_3 \mid z$ (11), by (2),(5),(9)

case $[\mathsf{E} = \mathsf{let}\ x = \mathsf{E}_1\ \mathsf{in}\ e_3$ and $e = \mathsf{let}\ x = e_2\ \mathsf{in}\ e_3]$.
$[\mathsf{H}]e_2 \longmapsto^* v$ (1), given
$e' = \mathsf{let}\ x = v\ \mathsf{in}\ e_3$ (2), given
$\mathsf{H} \mid e_2 \longmapsto^* \mathsf{H}' \mid y$ (3), by induction
$[\mathsf{H}']y = v$ (4), by induction
$[\mathsf{H}]e' \longmapsto^* w$ (5), by induction
$\mathsf{H}' \mid (\mathsf{let}\ x = y\ \mathsf{in}\ e_3) \longmapsto^* \mathsf{H}'' \mid z$ (6), by induction
$[\mathsf{H}'']z = w$ (7), by induction
$\mathsf{H} \mid (\mathsf{let}\ x = e_2\ \mathsf{in}\ e_3) \longmapsto^* \mathsf{H}'' \mid z$ (8), by (3),(4)

case $[\mathsf{E} = \mathsf{match}\ \mathsf{E}_1\ \{\overline{p_i \to e_i}\}$ and $e = \mathsf{match}\ e_2\ \{\overline{p_i \to e_i}\}]$.
$[\mathsf{H}]e_2 \longmapsto^* v$ (1), given
$e' = \mathsf{match}\ v\ \{\overline{p_i \to e_i}\}$ (2), given
$\mathsf{H} \mid e_2 \longmapsto^* \mathsf{H}' \mid y$ (3), by induction
$[\mathsf{H}']y = v$ (4), by induction
$[\mathsf{H}]e' \longmapsto^* w$ (5), by induction
$\mathsf{H}' \mid (\mathsf{match}\ v\ \{\overline{p_i \to e_i}\}) \longmapsto^* \mathsf{H}'' \mid z$ (6), by induction
$[\mathsf{H}'']z = w$ (7), by induction
$\mathsf{H} \mid (\mathsf{match}\ e_2\ \{\overline{p_i \to e_i}\}) \longmapsto^* \mathsf{H}'' \mid z$ (8), by (3),(4)

D.3 A Heap Reference Counting Calculus

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

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

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

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

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

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

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

$$\frac{\mathsf{H},\ x \mapsto^{n}\ v,\ r \mapsto^{1}\ () \vdash\ e\ \dashv\ \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{n+1}\ v \vdash\ r \leftarrow \mathsf{dropru}\ x;\ e\ \dashv\ \mathsf{H}_1}\ [\textsc{drdropru}]$$

$$\frac{\mathsf{H},\ r \mapsto^{1}\ () \vdash\ \mathsf{drop}\ \overline{y};\ e\ \dashv\ \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1}\ C\ \overline{y} \vdash\ r \leftarrow \mathsf{dropru}\ x;\ e\ \dashv\ \mathsf{H}_1}\ [\textsc{drdropconru}]$$

$$\frac{\mathsf{H},\ r \mapsto^{1}\ () \vdash\ \mathsf{drop}\ \overline{y};\ e\ \dashv\ \mathsf{H}_1}{\mathsf{H},\ x \mapsto^{1}\ \lambda_{\overline{y}} z.\ e \vdash\ r \leftarrow \mathsf{dropru}\ x;\ e\ \dashv\ \mathsf{H}_1}\ [\textsc{drdroplamru}]$$

$$\frac{\mathsf{H} \vdash\ x_1\ \dashv\ \mathsf{H}_1 \ldots \mathsf{H}_{n-1} \vdash\ x_n\ \dashv\ \mathsf{H}_n}{\mathsf{H} \vdash\ C\ x_1 \ldots x_n\ \dashv\ \mathsf{H}_n}\ [\textsc{drcon}]$$

$$\frac{\mathsf{H} \vdash\ \overline{y}\ \dashv\ \mathsf{H}_1 \qquad \overline{y} \mapsto^{1}\ (),\ x \mapsto^{1}\ () \vdash\ e\ \dashv\ \varnothing}{\mathsf{H} \vdash\ \lambda_{\overline{y}}\ x.\ e\ \dashv\ \mathsf{H}_1}\ [\textsc{drlam}]$$

$$\frac{\mathsf{H} \vdash\ e\ \dashv\ \mathsf{H}_1 \qquad \mathsf{H}_1 \vdash\ x\ \dashv\ \mathsf{H}_2}{\mathsf{H} \vdash\ e\ x\ \dashv\ \mathsf{H}_2}\ [\textsc{drapp}]$$

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

$$\frac{\mathsf{H},\ [\![\overline{z_i}]\!] \vdash\ e_i\ \dashv\ \mathsf{H}' \qquad \overline{z_i} \notin \mathsf{H}, \mathsf{H}' \qquad x\ \in \mathsf{H}}{\mathsf{H} \vdash\ \mathsf{match}\ x\ \{\ \overline{p_i\ \mapsto \mathsf{dup}(\overline{z_i});\ e_i}\ \}\ \dashv\ \mathsf{H}'}\ [\textsc{drmatch}]$$

For the next sections we will assume that $(\star)$ = `true`. Note however, that this is only relevant in lemma 12 and lemma 14. All other lemmas work for any choice of $(\star)$ (even `false`) as they either don't need the DRBIND rule or already assume $(\star)$ as a premise.

Lemma 5. (*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 5*) By a straightforward induction on the rules. □

Definition 4. (*Extension*)
H is extended with x, denoted as $\mathsf{H} +\!\!\!+\ x$, where $+\!\!\!+$ works as follows:
(1) if $\mathsf{H}\ =\ \mathsf{H}',\ x \mapsto^{n}\ v$, then $\mathsf{H} +\!\!\!+\ x\ =\ \mathsf{H}',\ x \mapsto^{n+1}\ v$;
(2) if $x \notin \mathsf{H}$, then $\mathsf{H} +\!\!\!+\ x\ =\ (\mathsf{H},\ x \mapsto^{1}\ v) +\!\!\!+\ \mathsf{fv}(v)$.
We omit the domain of x in $\mathsf{H} +\!\!\!+\ x$ for simplicity. The domain should always be available by inspecting the heap (in (1)) or via explicit passing (in (2)).

We only focus on situations where there is no cycles in the dependency of x (but we are fine with existing cycles in H), so that the extension terminates. That implies $(\mathsf{H},\ x \mapsto^1 v) \mathbin{+\!\!+} \mathsf{fv}(v) = \mathsf{H} \mathbin{+\!\!+} \mathsf{fv}(v),\ x \mapsto^1 v$ in (2).

Lemma 6. (*Drop is dual to extension*)
If $\mathsf{H}_1 \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}_2$, then $\mathsf{H}_1 = \mathsf{H}_2 \mathbin{+\!\!+} x$. Similarly, if $\mathsf{H}_1 \vdash x \dashv \mathsf{H}_2$, then $\mathsf{H}_1 = \mathsf{H}_2 \mathbin{+\!\!+} x$.

Proof. (*Of Lemma 6*) By induction on the judgment.
case

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

case

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

case

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

□

Lemma 7. (*Extension is dual to drop*)
$\mathsf{H} \mathbin{+\!\!+} x \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}$. Similarly, $\mathsf{H} \mathbin{+\!\!+} x \vdash x \dashv \mathsf{H}$.

Proof. (*Of Lemma 7*) By induction on $\mathbin{+\!\!+} x$.
case

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

case

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

□

Lemma 8. (*Extension Commutativity*)
$\mathsf{H} \mathbin{+\!\!+} x \mathbin{+\!\!+} y = \mathsf{H} \mathbin{+\!\!+} y \mathbin{+\!\!+} x$.

Proof. (*Of Lemma 8*) By induction on $\mathbin{+\!\!+} x$ and $\mathbin{+\!\!+} y$, then we do case analysis. **case** $x \in \mathsf{H}$. Then $\mathsf{H} = \mathsf{H}',\ x \mapsto^n v$.
By definition, $\mathsf{H} \mathbin{+\!\!+} x \mathbin{+\!\!+} y = (\mathsf{H}', x \mapsto^{n+1} v) \mathbin{+\!\!+} y$. Since the way $\mathbin{+\!\!+}$ works only depends on whether x exists but not the exact number of its occurrence, we can decrease the number of x, do $\mathbin{+\!\!+} y$ and then add x back. That is, $(\mathsf{H}', x \mapsto^{n+1} v) \mathbin{+\!\!+} y = (\mathsf{H}',\ x \mapsto^n v) \mathbin{+\!\!+} y \mathbin{+\!\!+} x = \mathsf{H} \mathbin{+\!\!+} y \mathbin{+\!\!+} x$.

case $y \in \mathsf{H}$ is similar as the previous case.
case $x,\ y \notin \mathsf{H}$. Then $\mathsf{H} \mathbin{+\!\!+} x = \mathsf{H} \mathbin{+\!\!+} \overline{x},\ x \mapsto^1 v$ where $\overline{x} = \mathsf{fv}(v)$.
subcase Assume $\mathbin{+\!\!+} y$ won't cause $\mathbin{+\!\!+} x$, then $\mathbin{+\!\!+} y$ doesn't care about the existance of x.
So $(\mathsf{H} \mathbin{+\!\!+} \overline{x},\ x \mapsto^1 v) \mathbin{+\!\!+} y = \mathsf{H} \mathbin{+\!\!+} \overline{x} \mathbin{+\!\!+} y,\ x \mapsto^1 v$
$= \mathsf{H} \mathbin{+\!\!+} y \mathbin{+\!\!+} \overline{x},\ x \mapsto^1 v$ by I.H.
$= \mathsf{H} \mathbin{+\!\!+} y \mathbin{+\!\!+} x$ by definition.
subcase Or otherwise $\mathbin{+\!\!+} y$ will cause $\mathbin{+\!\!+} x$. Since there is no cycle in the dependency, that means $\mathbin{+\!\!+} x$ won't cause $\mathbin{+\!\!+} y$. Then we can prove it as in the previous case. □

Lemma 9. (*Dropru is dual to extension*)
If $\mathsf{H}_1 \vdash r \leftarrow \mathsf{dropru}\ x;\ () \dashv \mathsf{H}_2$, then $\mathsf{H}_1 \mathbin{+\!\!+} r = \mathsf{H}_2 \mathbin{+\!\!+} x$.

Proof. (*Of Lemma 9*) **case**

$\mathsf{H},\ x \mapsto^{n+1} v \vdash r \leftarrow \mathsf{dropru}\ x;\ () \dashv \mathsf{H},\ x \mapsto^{n} v,\ r \mapsto^1 ()$	DRDROPRU
$\mathsf{H}, x \mapsto^{n+1} v \mathbin{+\!\!+} r = \mathsf{H}, x \mapsto^{n} v, r \mapsto^1 () \mathbin{+\!\!+} x$	Lemma 8

case

$\mathsf{H},\ x \mapsto^1 \lambda_{\overline{y}} z.\ e \vdash r \leftarrow \mathsf{dropru}\ x;\ () \dashv \mathsf{H}_1,\ r \mapsto^1 ()$	given
$\mathsf{H} \vdash \mathsf{drop}\ \overline{y};\ () \dashv \mathsf{H}_1$	DRDROPLAMRU
$\mathsf{H} = \mathsf{H}_1 \mathbin{+\!\!+} \overline{y}$	Lemma 6
$\mathsf{H}, x \mapsto^1 \lambda_{\overline{y}} z.e = \mathsf{H}_1 \mathbin{+\!\!+} x$	by definition
$\mathsf{H}, x \mapsto^1 \lambda_{\overline{y}} z.e \mathbin{+\!\!+} r = \mathsf{H}_1, r \mapsto^1 () \mathbin{+\!\!+} x$	Lemma 8

case

$\mathsf{H},\ x \mapsto^1 C\ \overline{y} \vdash r \leftarrow \mathsf{dropru}\ x;\ () \dashv \mathsf{H}_1,\ r \mapsto^1 ()$	given
$\mathsf{H} \vdash \mathsf{drop}\ \overline{y};\ () \dashv \mathsf{H}_1$	DRDROPCONRU
$\mathsf{H} \mathbin{+\!\!+} r = \mathsf{H}_1 \mathbin{+\!\!+} \overline{y}$	Lemma 6
$\mathsf{H}, x \mapsto^1 C\ \overline{y} = \mathsf{H}_1 \mathbin{+\!\!+} x$	by definition
$\mathsf{H}, x \mapsto^1 C\ \overline{y} \mathbin{+\!\!+} r = \mathsf{H}_1, r \mapsto^1 () \mathbin{+\!\!+} x$	Lemma 8

□

Lemma 10. (*Extension is dual to dropru*)
$\mathsf{H} \mathbin{+\!\!+} x \vdash r \leftarrow \mathsf{dropru}\ x;\ () \dashv \mathsf{H} \mathbin{+\!\!+} r$.

Proof. (*Of Lemma 10*) By induction on $\mathbin{+\!\!+} x$.
case

$\mathsf{H} = \mathsf{H}',\ x \mapsto^{n} \overline{y}$	if
$\mathsf{H} \mathbin{+\!\!+} x = \mathsf{H}',\ x \mapsto^{n+1} \overline{y}$	by definition
$\mathsf{H}',\ x \mapsto^{n+1} \overline{y} \vdash r \leftarrow \mathsf{dropru}\ x;\ () \dashv \mathsf{H},\ r \mapsto^1 ()$	DRDROPRU

case

$x \notin \mathsf{H}$	if
$\overline{y} = \mathsf{fv}(v)$	let
$\mathsf{H} \mathbin{+\!\!+} x = \mathsf{H} \mathbin{+\!\!+} \overline{y},\ x \mapsto^1 v$	by definition
$\mathsf{H} \mathbin{+\!\!+} \overline{y} \vdash \overline{y} \dashv \mathsf{H}$	I.H.
$\mathsf{H} \mathbin{+\!\!+} \overline{y},\ x \mapsto^1 v \vdash r \leftarrow \mathsf{dropru}\ x;\ () \dashv \mathsf{H},\ r \mapsto^1 ()$	DRDROPLAMRU or DRDROPCONRU

□

D.3.1 Relating to linear resource calculus.

Definition 5. (*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 11.
$[\![\Gamma,\ x]\!] \vdash x \dashv [\![\Gamma]\!]$.
Similarly, if $[\![\Gamma]\!] \vdash e \dashv \mathsf{H}$, then $[\![\Gamma, x]\!] \vdash \mathsf{drop}\ x;\ e \dashv \mathsf{H}$.
Similarly, if $[\![\Gamma, r]\!] \vdash e \dashv \mathsf{H}$, then $[\![\Gamma, x]\!] \vdash r \leftarrow \mathsf{dropru}\ x;\ e \dashv \mathsf{H}$.

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

Lemma 12. (*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 12*) By induction on the elaboration.
case

$\Delta \mid x \vdash x \rightsquigarrow x$	given
$[\![\Delta,\ x]\!] \vdash x \dashv [\![\Delta]\!]$	Lemma 11

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 11

case

$\Delta \mid \Gamma, x \vdash e \rightsquigarrow r \leftarrow \mathsf{dropru}\ x;\ e'$	given
$\Delta \mid \Gamma, r \vdash e \rightsquigarrow e'$	given
$[\![\Delta, \Gamma, r]\!] \vdash e' \dashv [\![\Delta]\!]$	I.H.
$[\![\Delta, \Gamma, x]\!] \vdash r \leftarrow \mathsf{dropru}\ x;\ e' \dashv [\![\Delta]\!]$	Lemma 11

case

$\Delta \mid \overline{y} \vdash \lambda x.\ e \rightsquigarrow \lambda_{\overline{y}}\ x.\ e'$	given
$\varnothing \mid \overline{y}, x \vdash e \rightsquigarrow e'$	given
$\overline{y} = \mathsf{fv}(\lambda x.\ e)$	given
$[\![\Delta, \overline{y}]\!] \vdash \overline{y} \dashv [\![\Delta]\!]$	Lemma 11
$[\![\overline{y}, x]\!] \vdash e' \dashv \varnothing$	I.H.
$[\![\Delta, \overline{y}]\!] \vdash \lambda_{\overline{y}}\ x.\ e \dashv [\![\Delta]\!]$	DRLAM

case

$\Delta \mid \Gamma \vdash e_1\ x \ \rightsquigarrow\ e_1'\ x$	given
$\Delta, x \mid \Gamma \vdash e_1 \ \rightsquigarrow\ e_1'$	given
$[\![\Delta, x, \Gamma]\!] \vdash e_1' \ \dashv\ [\![\Delta,\ x]\!]$	I.H.
$[\![\Delta, x]\!] \vdash x \ \dashv\ [\![\Delta]\!]$	Lemma 11
$[\![\Delta, \Gamma]\!] \vdash e_1'\ x \ \dashv\ [\![\Delta]\!]$	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

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

case

$\Delta \mid \overline{x} \vdash C\ \overline{x} \ \rightsquigarrow\ C\ \overline{x}$	given
$[\![\Delta, \overline{x}]\!] \vdash \overline{x} \vdash [\![\Delta]\!]$	Lemma 11
$[\![\Delta, \overline{x}]\!] \vdash C\ \overline{x} \ \dashv\ [\![\Delta]\!]$	DRCON

□

D.3.2 Weakening.

Lemma 13. (*Weakening*)
If $\mathsf{H}_1 \vdash e \ \dashv\ \mathsf{H}_2$, then $\mathsf{H}_1 +\!\!\!+\ x \vdash e \ \dashv\ \mathsf{H}_2 +\!\!\!+\ x$.

Proof. (*Of Lemma 13*) By induction on the judgment.
case

$\mathsf{H}_0 \vdash C\ \overline{x} - \mid \mathsf{H}_n$	given
$\mathsf{H}_{i-1} \vdash x_i \ \dashv\ \mathsf{H}_i$	DRCON
$\mathsf{H}_{i-1} +\!\!\!+ x \vdash x_i \ \dashv\ \mathsf{H}_i +\!\!\!+ x$	I.H.
$\mathsf{H}_0 +\!\!\!+ x \vdash C\ \overline{x} \ \dashv\ \mathsf{H}_n +\!\!\!+\ x$	DRCON

case

$\mathsf{H} \vdash y \ \dashv\ \mathsf{H}_2$	given
$\mathsf{H}\ =\ \mathsf{H}_2 +\!\!\!+\ y$	Lemma 6
$\mathsf{H} +\!\!\!+\ x\ =\ \mathsf{H}_2 +\!\!\!+\ y +\!\!\!+\ x$	
$=\ \mathsf{H}_2 +\!\!\!+\ x +\!\!\!+\ y$	Lemma 8
$\mathsf{H}_2 +\!\!\!+\ x +\!\!\!+\ y \vdash y \ \dashv\ \mathsf{H}_2 +\!\!\!+\ x$	Lemma 7

case

$\mathsf{H},\ y \mapsto^{n} \overline{y} \vdash \mathsf{dup}\ y;\ e \dashv \mathsf{H}_1$	given
$\mathsf{H},\ y \mapsto^{n+1} \overline{y} \vdash e \dashv \mathsf{H}_1$	DRDUP
$(\mathsf{H},\ y \mapsto^{n} \overline{y}) \mathbin{+\!\!+} y \vdash e \dashv \mathsf{H}_1$	definition of $\mathbin{+\!\!+}$
$(\mathsf{H},\ y \mapsto^{n} \overline{y}) \mathbin{+\!\!+} y \mathbin{+\!\!+} x \vdash e \dashv \mathsf{H}_1 \mathbin{+\!\!+} x$	I.H.
$(\mathsf{H},\ y \mapsto^{n} \overline{y}) \mathbin{+\!\!+} y \mathbin{+\!\!+} x$	
$= (\mathsf{H},\ y \mapsto^{n} \overline{y}) \mathbin{+\!\!+} x \mathbin{+\!\!+} y$	Lemma 8
$(\mathsf{H},\ y \mapsto^{n} \overline{y}) \mathbin{+\!\!+} x \mathbin{+\!\!+} y \vdash e \dashv \mathsf{H}_1 \mathbin{+\!\!+} x$	By substitution
$(\mathsf{H},\ y \mapsto^{n} \overline{y}) \mathbin{+\!\!+} x \vdash \mathsf{dup}\ y;\ e \dashv \mathsf{H}_1 \mathbin{+\!\!+} x$	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
$\mathsf{H} = \mathsf{H}_3 \mathbin{+\!\!+} y$	Lemma 6
$\mathsf{H} \mathbin{+\!\!+} x = \mathsf{H}_3 \mathbin{+\!\!+} y \mathbin{+\!\!+} x$	
$= \mathsf{H}_3 \mathbin{+\!\!+} x \mathbin{+\!\!+} y$	Lemma 8
$\mathsf{H}_3 \mathbin{+\!\!+} x \mathbin{+\!\!+} y \vdash \mathsf{drop}\ y;\ () \dashv \mathsf{H}_3 \mathbin{+\!\!+} x$	Lemma 7
$\mathsf{H}_3 \mathbin{+\!\!+} x \vdash e \dashv \mathsf{H}_2 \mathbin{+\!\!+} x$	I.H.
$\mathsf{H}_3 \mathbin{+\!\!+} x \mathbin{+\!\!+} y \vdash \mathsf{drop}\ y;\ e \dashv \mathsf{H}_2 \mathbin{+\!\!+} x$	Follows
$\mathsf{H} \mathbin{+\!\!+} x \vdash \mathsf{drop}\ y;\ e \dashv \mathsf{H}_2 \mathbin{+\!\!+} x$	By substitution

case

$\mathsf{H} \vdash r \leftarrow \mathsf{dropru}\ y;\ e \dashv \mathsf{H}_2$	given
$\mathsf{H} \vdash \mathsf{drop}\ y;\ () \dashv \mathsf{H}_3$	follows
$\mathsf{H}_3 \mathbin{+\!\!+} r \vdash e \dashv \mathsf{H}_2$	above
$\mathsf{H} = \mathsf{H}_3 \mathbin{+\!\!+} y$	Lemma 6
$\mathsf{H} \mathbin{+\!\!+} x = \mathsf{H}_3 \mathbin{+\!\!+} y \mathbin{+\!\!+} x$	
$= \mathsf{H}_3 \mathbin{+\!\!+} x \mathbin{+\!\!+} y$	Lemma 8
$\mathsf{H}_3 \mathbin{+\!\!+} x \mathbin{+\!\!+} y \vdash r \leftarrow \mathsf{dropru}\ y;\ () \dashv \mathsf{H}_3 \mathbin{+\!\!+} x \mathbin{+\!\!+} r$	Lemma 10
$\mathsf{H}_3 \mathbin{+\!\!+} r \mathbin{+\!\!+} x \vdash e \dashv \mathsf{H}_2 \mathbin{+\!\!+} x$	I.H.
$\mathsf{H}_3 \mathbin{+\!\!+} x \mathbin{+\!\!+} r = \mathsf{H}_3 \mathbin{+\!\!+} r \mathbin{+\!\!+} x$	Lemma 8
$\mathsf{H}_3 \mathbin{+\!\!+} x \mathbin{+\!\!+} y \vdash r \leftarrow \mathsf{dropru}\ y;\ e \dashv \mathsf{H}_2 \mathbin{+\!\!+} x$	Follows
$\mathsf{H} \mathbin{+\!\!+} x \vdash r \leftarrow \mathsf{dropru}\ y;\ e \dashv \mathsf{H}_2 \mathbin{+\!\!+} x$	By substitution

case

$\mathsf{H} \vdash \lambda_{\overline{y}}\, z.\, e \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash \overline{y} \dashv \mathsf{H}_1$	given
$\overline{y} \mapsto^{1} (),\ z \mapsto^{1} () \vdash e \dashv \varnothing$	given
$\mathsf{H} \mathbin{+\!\!+} x \vdash \overline{y} \dashv \mathsf{H}_1 \mathbin{+\!\!+} x$	I.H.
$\mathsf{H} \mathbin{+\!\!+} x \vdash \lambda_{\overline{y}}\, z.\, e \dashv \mathsf{H}_1 \mathbin{+\!\!+} x$	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
$\mathsf{H} \mathbin{+\!\!+} x \vdash e_1 \dashv \mathsf{H}_1 \mathbin{+\!\!+} x$	I.H.
$\mathsf{H}_1 \mathbin{+\!\!+} x \vdash e_2 \dashv \mathsf{H}_2 \mathbin{+\!\!+} x$	I.H.
$\mathsf{H} \mathbin{+\!\!+} x \vdash e_1\ e_2 \dashv \mathsf{H}_2 \mathbin{+\!\!+} x$	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
$\mathsf{H} \mathbin{+\!\!+} x \vdash e_1\ \dashv\ \mathsf{H}_1 \mathbin{+\!\!+} x$	I.H.
$(\mathsf{H}_1,\ z \mapsto^1 ()) \mathbin{+\!\!+} x \vdash e_2\ \dashv\ \mathsf{H}_2 \mathbin{+\!\!+} x$	I.H.
$z \notin \mathsf{H}_1$	Lemma 5
$\mathsf{H}_1, z \mapsto^1 ()\ =\ \mathsf{H}_1 \mathbin{+\!\!+} z$	
$(\mathsf{H}_1,\ z \mapsto^1 ()) \mathbin{+\!\!+} x\ =\ \mathsf{H}_1 \mathbin{+\!\!+} z \mathbin{+\!\!+} x$	
$=\ \mathsf{H}_1 \mathbin{+\!\!+} x \mathbin{+\!\!+} z$	Lemma 8
$=\ \mathsf{H}_1 \mathbin{+\!\!+} x,\ z \mapsto^1 ()$	
$\mathsf{H}_1 \mathbin{+\!\!+} x,\ z \mapsto^1 () \vdash e_2\ \dashv\ \mathsf{H}_2 \mathbin{+\!\!+} x$	by substitution
$\mathsf{H} \mathbin{+\!\!+} x \vdash e_1\ e_2\ \dashv\ \mathsf{H}_2 \mathbin{+\!\!+} x$	DRBIND

case

$\mathsf{H} \vdash \mathsf{match}\ z\ \{\ \overline{p_i\ \mapsto \mathsf{dup}(\overline{z_i});\ e_i}\ \}\ \dashv\ \mathsf{H}'$	given
$z\ \in \mathsf{H},\ z\ \in \mathsf{H} \mathbin{+\!\!+} x$	given
$\mathsf{H}, \mathsf{H}_i \vdash e_i\ \dashv\ \mathsf{H}'$	given
$\mathsf{H}_i\ =\ \llbracket \overline{z_i} \rrbracket$	given
$(\mathsf{H}, \mathsf{H}_i) \mathbin{+\!\!+} x \vdash e_i\ \dashv\ \mathsf{H}' \mathbin{+\!\!+} x$	I.H.
$\overline{z_i}$ fresh	assume
$\mathsf{H},\ \llbracket \mathsf{H}_i \rrbracket\ =\ \mathsf{H} \mathbin{+\!\!+} \overline{z_i}$	
$(\mathsf{H},\ \mathsf{H}_i) \mathbin{+\!\!+} x\ =\ \mathsf{H} \mathbin{+\!\!+} \overline{z_i} \mathbin{+\!\!+} x$	
$=\ \mathsf{H} \mathbin{+\!\!+} x \mathbin{+\!\!+} \overline{z_i}$	Lemma 8
$=\ \mathsf{H} \mathbin{+\!\!+} x,\ \mathsf{H}_i$	
$\mathsf{H} \mathbin{+\!\!+} x, \mathsf{H}_i \vdash e_i\ \dashv\ \mathsf{H}' \mathbin{+\!\!+} x$	by substitution
$\mathsf{H} \mathbin{+\!\!+} x \vdash \mathsf{match}\ z\ \{\ \overline{p_i\ \mapsto \mathsf{dup}(\overline{z_i});\ e_i}\ \}\ \dashv\ \mathsf{H}' \mathbin{+\!\!+} x$	DRMATCH

□

D.3.3 Substitution.

Lemma 14. (*Substitution preserves dependencies*)
If $y, z \notin \mathsf{H}_1, \mathsf{H}_2$ and $\mathsf{H}_1, \llbracket y \rrbracket \vdash e\ \dashv\ \mathsf{H}_2$, then $\mathsf{H}_1\ \mathbin{+\!\!+}\ z \vdash e[y{:=}z]\ \dashv\ \mathsf{H}_2$.

Proof. (*Of Lemma 14*) By induction on the judgment. The cases DRVAR, DRVARLAM, DRVARCON, DRDROP, DRDROPCON, DRDROPLAM, DRDROPRU, DRDROPCONRU, and DRDROPLAMRU follow directly from lemma 6 (resp. 9) if $y\ =\ x$. If $y \neq x$, then the derivation encountered a var, drop or drop-reuse case with $y\ =\ x$. The claim then follows by the inductive hypothesis.

case

$\mathsf{H}_0 \vdash\ C\ \overline{x} - \mid \mathsf{H}_n$	given
$\mathsf{H}_{i-1} \vdash x_i\ \dashv\ \mathsf{H}_i$	DRCON
$y\ \notin \mathsf{H}_n$	given
$\exists j.\ y \in \mathsf{H}_j,\ \ y \notin \mathsf{H}_{j+1}$	follows
$(\mathsf{H}_i\ -\ \llbracket y \rrbracket){+\!\!+}z \vdash x_i[y{:=}z]\ \dashv\ (\mathsf{H}_{i+1}\ -\ \llbracket y \rrbracket){+\!\!+}z$	for $i\ \langle\ j$, by Lemma 13
$(\mathsf{H}_j\ -\ \llbracket y \rrbracket){+\!\!+}z \vdash y[y{:=}z]\ \dashv\ \mathsf{H}_{j+1}$	I.H.
$\mathsf{H}_i \vdash x_i[y{:=}z]\ \dashv\ \mathsf{H}_{i+1}$	for $i\ \rangle\ j$, above
$(\mathsf{H}_0\ -\ \llbracket y \rrbracket) \mathbin{+\!\!+} z \vdash (C\ \overline{x})[y{:=}z]\ \dashv\ \mathsf{H}_n$	DRCON

case

$$
\begin{array}{ll}
\mathsf{H}, [\![y]\!] \vdash \mathsf{dup}\ x;\ e \dashv \mathsf{H}_1 & \text{given} \\
(\mathsf{H} \mathbin{+\!\!\!+} x), [\![y]\!] \vdash e \dashv \mathsf{H}_1 & \text{by lemma 8} \\
(\mathsf{H} \mathbin{+\!\!\!+} x) \mathbin{+\!\!\!+} z \vdash e[y{:=}z] \dashv \mathsf{H}_1 & \text{I.H.} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash \mathsf{dup}\ x[y{:=}z];\ e[y{:=}z] \dashv \mathsf{H}_1 & \textsc{drdup} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash (\mathsf{dup}\ x;\ e)[y{:=}z] \dashv \mathsf{H}_1 & \text{follows}
\end{array}
$$

case

$$
\begin{array}{ll}
\mathsf{H}, [\![y]\!] \vdash \lambda_{\overline{y}}\, x.\ e \dashv \mathsf{H}_1 & \text{given} \\
\mathsf{H}, [\![y]\!] \vdash \overline{y} \dashv \mathsf{H}_1 & \text{given} \\
\overline{y} \mapsto^1 (),\ x \mapsto^1 () \vdash e \dashv \varnothing & \text{given} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash \overline{y}[y{:=}z] \dashv \mathsf{H}_1 & \text{I.H.} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash (\lambda_{\overline{y}}\, x.\ e)[y{:=}z] \dashv \mathsf{H}_1 \mathbin{+\!\!\!+} z & \textsc{drlam}
\end{array}
$$

case

$$
\begin{array}{ll}
\mathsf{H}, [\![y]\!] \vdash e_1\ x \dashv \mathsf{H}_2 & \text{given} \\
\mathsf{H}, [\![y]\!] \vdash e_1 \dashv \mathsf{H}_1 & \text{given} \\
y \notin \mathsf{H}_1 & \text{assume} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash e_1[y{:=}z] \dashv \mathsf{H}_1 & \text{I.H.} \\
\mathsf{H}_1 \vdash x \dashv \mathsf{H}_2 & \text{given} \\
\mathsf{H}_1 \vdash x[y{:=}z] \dashv \mathsf{H}_2 & \text{since } x{\neq}y \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash (e_1\ x)[y{:=}z] \dashv \mathsf{H}_2 & \textsc{drapp} \\
y \in \mathsf{H}_1 & \text{else} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash e_1[y{:=}z] \dashv (\mathsf{H}_1 - [\![y]\!]) \mathbin{+\!\!\!+} z & \text{by substitution and Lemma 13} \\
(\mathsf{H}_1 - [\![y]\!]) \mathbin{+\!\!\!+} z \vdash x[y{:=}z] \dashv \mathsf{H}_2 & \text{I.H.} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash (e_1\ x)[y{:=}z] \dashv \mathsf{H}_2 & \textsc{drapp}
\end{array}
$$

case

$$
\begin{array}{ll}
\mathsf{H}, [\![y]\!] \vdash \mathsf{val}\ x = e_1;\ e_2 \dashv \mathsf{H}_2 & \text{given} \\
\mathsf{H}, [\![y]\!] \vdash e_1 \dashv \mathsf{H}_1 & \text{given} \\
y \notin \mathsf{H}_1 & \text{assume} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash e_1[y{:=}z] \dashv \mathsf{H}_1 & \text{I.H.} \\
\mathsf{H}_1, [\![x]\!] \vdash e_2 \dashv \mathsf{H}_2 & \text{given} \\
\mathsf{H}_1, [\![x]\!] \vdash e_2[y{:=}z] \dashv \mathsf{H}_2 & \text{since } y \notin \mathsf{fv}(e_2) \text{ by lemma 5} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash (\mathsf{val}\ x = e_1;\ e_2)[y{:=}z] \dashv \mathsf{H}_2 & \textsc{drbind} \\
y \in \mathsf{H}_1 & \text{else} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash e_1[y{:=}z] \dashv (\mathsf{H}_1 - [\![y]\!]) \mathbin{+\!\!\!+} z & \text{by substitution and Lemma 13} \\
(\mathsf{H}_1 - [\![y]\!], [\![x]\!]) \mathbin{+\!\!\!+} z \vdash e_2[y{:=}z] \dashv \mathsf{H}_2 & \text{I.H.} \\
\mathsf{H} \mathbin{+\!\!\!+} z \vdash (\mathsf{val}\ x = e_1;\ e_2)[y{:=}z] \dashv \mathsf{H}_2 & \textsc{drbind}
\end{array}
$$

case

$\mathsf{H} \vdash \mathsf{match}\ x\ \{\ \overline{p_i\ \mapsto \mathsf{dup}(\overline{z_i});\ e_i}\ \}\ \dashv\ \mathsf{H}'$	given
$x\ \in \mathsf{H},\ x[y{:=}z]\ \in (\mathsf{H}\ -\ [\![y]\!]) + z$	given
$\mathsf{H}, \mathsf{H}_i \vdash\ e_i\ \dashv\ \mathsf{H}'$	given
$\mathsf{H}_i\ =\ [\![\overline{z_i}]\!]$	given
$((\mathsf{H}, \mathsf{H}_i)\ -\ [\![y]\!]) + z \vdash\ e_i[y{:=}z]\ \dashv\ \mathsf{H}'$	I.H.
$\overline{z_i}$ fresh	assume
$\mathsf{H},\ [\![\mathsf{H}_i]\!]\ =\ \mathsf{H} + \overline{z_i}$	
$((\mathsf{H},\ \mathsf{H}_i)\ -\ [\![y]\!]) + z\ =\ (\mathsf{H}\ -\ [\![y]\!]) + \overline{z_i} + z$	
$=\ (\mathsf{H}\ -\ [\![y]\!]) + z + \overline{z_i}$	Lemma 8
$=\ (\mathsf{H}\ -\ [\![y]\!]) + z,\ \mathsf{H}_i$	
$(\mathsf{H}\ -\ [\![y]\!]) + z, \mathsf{H}_i \vdash\ e_i\ \dashv\ \mathsf{H}'$	by substitution
$(\mathsf{H}\ -\ [\![y]\!]) + z \vdash\ (\mathsf{match}\ x\ \{\ \overline{p_i\ \mapsto \mathsf{dup}(\overline{z_i});\ e_i}\ \})[y{:=}z]\ \dashv\ \mathsf{H}'$	DRMATCH

□

D.4 Soundness of Reference Counting Semantics

Definition 6. (*Well-formed Abstractions*)
If e ok, then all $(\lambda^{\overline{y}}\ x.e_1)$ in e satisfies $[\![\overline{y},\ x]\!] \vdash\ e_1\ \dashv\ \varnothing$.

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; (2) there are no dependency cycles in H.

Lemma 15. (*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 15*) When we place a new variable $z \mapsto^1\ v$ in the heap (e.g., (lam)), z is fresh so v 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 5, 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 $[\![\overline{y},\ x]\!] \vdash\ e\ \dashv\ \varnothing$, then $[\![\overline{y}[y{:=}z],\ x]\!] \vdash\ e[y{:=}z]\ \dashv\ \varnothing$ by substitution.

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

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

$\mathsf{H} \vdash f\ z\ \dashv\ \mathsf{H}_1$	given
$\mathsf{H}\ =\ \mathsf{H}_1 + f + z$	Lemma 6
$\overline{y}\ \in \mathsf{H}_1 + f + z$	$f\ \mapsto\ \lambda_{\overline{y}}x.\ e$
$\mathsf{H} \vdash \mathsf{dup}\ \overline{y};\ ()\ \dashv\ \mathsf{H} + \overline{y}$	by definition
$\mathsf{H} + \overline{y}\ =\ \mathsf{H}_1 + f + z + \overline{y}$	
$=\ \mathsf{H}_1 + \overline{y} + z + f$	Lemma 8
$\mathsf{H}_1 + \overline{y} + z + f \vdash \mathsf{drop}\ f;\ ()\ \dashv\ \mathsf{H}_1 + \overline{y} + z$	Lemma 7
$[\![\overline{y},\ x]\!] \vdash\ e\ \dashv\ \varnothing$	$\lambda_{\overline{y}}x.\ e$ ok
$[\![\overline{y},\ z]\!] \vdash\ e[x{:=}z]\ \dashv\ \varnothing$	by substitution
$\mathsf{H}_1 + \overline{y} + z \vdash\ e[x{:=}z]\ \dashv\ \mathsf{H}_1$	Lemma 13 and Lemma 14

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

$\mathsf{H} \vdash \mathsf{match}\ x\ \{\overline{C\ \overline{x} \to \mathsf{dup}\ \overline{x};\ e_i}\}\ \dashv\ \mathsf{H}'$ given
$\mathsf{H}, [\![\overline{x}]\!] \vdash\ e_i\ \dashv\ \mathsf{H}'$ given
$\overline{x} \notin \mathsf{H}_i$ given
$(x \mapsto^n\ C\ \overline{y})\ \in \mathsf{H}$ given
$\overline{y}\ \in \mathsf{H}$ H ok
$\mathsf{H} \vdash\ \mathsf{dup}\ \overline{y};\ ()\ \dashv\ \mathsf{H} \mathbin{+\!\!\!+} \overline{y}$ by definition
$\mathsf{H} \mathbin{+\!\!\!+} \overline{y} \vdash\ e_i[\overline{x}{:=}\overline{y}]\ \dashv\ \mathsf{H}'$ Lemma 14
$\mathsf{H} \vdash\ \mathsf{dup}(\overline{x});\ e_i[\overline{x}{:=}\overline{y}]\ \dashv\ \mathsf{H}'$ DUP

case (let_r) $\mathsf{H} \mid \mathsf{let}\ x\ =\ z\ \mathsf{in}\ e \longrightarrow_r \mathsf{H} \mid e[x{:=}z]$

$\mathsf{H} \vdash\ \mathsf{let}\ x\ =\ z\ \mathsf{in}\ e\ \dashv\ \mathsf{H}'$ given
$\mathsf{H} \vdash\ z\ \dashv\ \mathsf{H}_1$ given
$\mathsf{H}_1\ =\ \mathsf{H} \mathbin{+\!\!\!+} z$ Lemma 6
$\mathsf{H}_1, x \mapsto^1 () \vdash\ e\ \dashv\ \mathsf{H}'$ given
$\mathsf{H}_1 \mathbin{+\!\!\!+} z \vdash\ e[x{:=}z]\ \dashv\ \mathsf{H}'$ Lemma 14
$\mathsf{H} \vdash\ e[x{:=}z]\ \dashv\ \mathsf{H}'$

case (lam_r) $\mathsf{H} \mid (\lambda_{\overline{y}}\ x.\ e) \longrightarrow_h \mathsf{H}, f \mapsto^1 \lambda_{\overline{y}}\ x.\ e \mid f$ fresh f

$\mathsf{H} \vdash\ \lambda_{\overline{y}}\ x.\ e\ \dashv\ \mathsf{H}_1$ given
$\mathsf{H} \vdash\ \overline{y}\ \dashv\ \mathsf{H}_1$ given
$\mathsf{H},\ f \mapsto^1 \lambda_{\overline{y}}\ x.\ e \vdash\ f\ \dashv\ \mathsf{H}_1$ 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 $(dlam_r)$ $\mathsf{H}, x \mapsto^1 \lambda_{\overline{y}} z.e \mid\ \mathsf{drop}\ x;\ e \longrightarrow_r \mathsf{H} \mid \mathsf{drop}\ \overline{y};\ e$

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

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

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

case $(drop_{ru})$ $\mathsf{H}, x \mapsto^{n+1}\ v \mid\ r \leftarrow \mathsf{dropru}\ x;\ e \longrightarrow_r \mathsf{H}, x \mapsto^n\ v, z \mapsto^1 () \mid e[r{:=}z]$ fresh z

$\mathsf{H},\ x \mapsto^{n+1}\ v \vdash\ r \leftarrow \mathsf{dropru}\ x;\ e\ \dashv\ \mathsf{H}_1$ given
$\mathsf{H},\ x \mapsto^n\ v,\ r \mapsto^1 () \vdash\ e\ \dashv\ \mathsf{H}_1$ DRDROPRU
$\mathsf{H},\ x \mapsto^n\ v,\ z \mapsto^1 () \vdash\ e[r{:=}z]\ \dashv\ \mathsf{H}_1$ by substitution

case $(dlam_{ru})$ $\mathsf{H}, x \mapsto^1 \lambda_{\overline{x}} y.e' \mid\ r \leftarrow \mathsf{dropru}\ x;\ e \longrightarrow_r \mathsf{H}, z \mapsto^1 () \mid \mathsf{drop}\ \overline{x};\ e[r{:=}z]$ fresh z

$\mathsf{H}, x \mapsto^1 \lambda_{\overline{y}} z.e \vdash\ r \leftarrow \mathsf{dropru}\ x;\ e\ \dashv\ \mathsf{H}_1$ given
$\mathsf{H}, r \mapsto^1 () \vdash \mathsf{drop}\ \overline{y};\ e\ \dashv\ \mathsf{H}_1$ DRDROPLAMRU
$\mathsf{H}, z \mapsto^1 () \vdash \mathsf{drop}\ \overline{y};\ e[r{:=}z]\ \dashv\ \mathsf{H}_1$ by substitution

case $(dcon_{ru})$ $\mathsf{H}, x \mapsto^1\ C\ \overline{x} \mid\ r \leftarrow \mathsf{dropru}\ x;\ e \longrightarrow_r \mathsf{H}, \overline{z} \mapsto^1 (), z \mapsto^1\ C\ \overline{z} \mid \mathsf{drop}\ \overline{x};\ e[r{:=}z]$
fresh $z, \overline{z}$

$\mathsf{H}, x \mapsto^1 C\ \overline{y} \vdash r \leftarrow \mathsf{dropru}\ x;\ e\ \dashv \mathsf{H}_1$	given
$\mathsf{H}, r \mapsto^1 () \vdash \mathsf{drop}\ \overline{y};\ e\ \dashv \mathsf{H}_1$	DRDROPCONRU
$\mathsf{H}, \overline{z} \mapsto^1 (), z \mapsto^1 C\ \overline{z} \vdash \mathsf{drop}\ \overline{y};\ e[r{:=}z]\ \dashv \mathsf{H}_1$	by Lemma 14

□

The ok part reasoning of Lemma 15 can be easily generalized to big step. So from now on we will implicitly assume every expression and heap we discuss is ok.

We introduced the heap evaluation context in Figure 10 as:

$\mathsf{E} ::= \Box \mid \mathsf{E}\ e \mid x\ \mathsf{E}$
$\mid \quad \mathsf{let}\ x = \mathsf{E}\ \mathsf{in}\ e$
$\mid \quad \mathsf{match}\ \mathsf{E}\ \{\ \overline{p_i \to e_i}\ \}$
$\mid \quad C\ x_1 \ldots x_{i-1}\ \mathsf{E}\ v_{i+1} \ldots v_n$

But that was only necessary for terms, that hadn't been normalized (as we encountered them in lemma 4). But in the next proofs we will only apply the heap semantics to normalized terms: An expression e' such that $\Delta \mid \Gamma \vdash e \rightsquigarrow e'$ needs to be normalized and the heap semantics transform normalized terms to normalized terms since only variables and not values are substituted in. For the remainder of this section we will therefore work with the evaluation context:

$\mathsf{E} ::= \Box \mid \mathsf{E}\ x \mid \mathsf{val}\ x = \mathsf{E};\ e$

Lemma 16. (*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 16*) By induction on E.

case $\mathsf{E} = \Box$. Follows by Lemma 15.

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

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

□

The next lemma can't be stated with $[\mathsf{H}_1]e_1 = [\mathsf{H}_1']e_1$, because reuse tokens are handled differently in the heap calculi and thus we might have $\mathsf{H}_1 = r \mapsto^1 ()$, $\mathsf{H}_1' = r \mapsto^1 C\ \overline{v}$, $e_1 = r$. But we know that reuse tokens can only appear as the argument to dup/drop/dropru (as they are chosen fresh in the DROPRU rule), and so we can remove them beforehand ($\lceil e_1 \rceil$). Because of the (lam_h) rule, we denote by $\lceil e_1 \rceil$ here a procedure that deletes all dups, drops and droprus in e_1 but doesn't descend into lambdas (and doesn't delete dups, drops, droprus there).

Lemma 17. (*Reference counting semantics is sound (small step)*)
If $\mathsf{H}_1 \mid e_1 \longrightarrow_h \mathsf{H}_2 \mid e_2$, and $\mathsf{H}_1' \vdash e_1\ \dashv \mathsf{H}_3$, and $[\mathsf{H}_1]\lceil e_1 \rceil = [\mathsf{H}_1']\lceil e_1 \rceil$, then there exists H_2' such that $\mathsf{H}_1' \mid e_1 \longrightarrow^*{}_r \mathsf{H}_2' \mid e_2$ and $[\mathsf{H}_2]\lceil e_2 \rceil = [\mathsf{H}_2']\lceil e_2 \rceil$.

Proof. (*Of Lemma 17*) By case analysis on the heap judgment. The rules (lam_h), (con_h), (app_h), $(match_h)$, (let_h) and (lam_r), (con_r), (app_r), $(match_r)$, (let_r) are identical. Note in particular, that in the (lam_h) and (lam_r) rules we store the same lambda, since $\lceil e_1 \rceil$ doesn't descend into lambdas. We thus only have to show the claim for the (dup_h), $(drop_h)$ and $(dropru_h)$ rules.

case (dup_h) $\mathsf{H} \mid \mathsf{dup}\ x;\ e \longrightarrow_h \mathsf{H} \mid e$

$\mathsf{H}_1' \vdash \mathsf{dup}\ x;\ e \dashv \mathsf{H}_3$	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 $+\!\!+$
$\mathsf{H}_2' = \mathsf{H}_1' +\!\!+\, x$	follows
$[\mathsf{H}_1]\lceil e \rceil = [\mathsf{H}_1']\lceil e \rceil$	given
$[\mathsf{H}_2]\lceil e \rceil = [\mathsf{H}_1]\lceil e \rceil = [\mathsf{H}_1']\lceil e \rceil = [\mathsf{H}_2']\lceil e \rceil$	follows

case $(drop_h)$ $\mathsf{H} \mid \mathsf{drop}\ x;\ e \longrightarrow_h \mathsf{H} \mid e$

$\mathsf{H}_1' \vdash \mathsf{drop}\ x;\ e \dashv \mathsf{H}_3$	given
$\mathsf{H}_1' \vdash \mathsf{drop}\ x;\ () \dashv \mathsf{H}_2'$	follows
$\mathsf{H}_2' \vdash e \dashv \mathsf{H}_3$	above
$\mathsf{fv}(e) \in \mathsf{H}_2'$	Lemma 5
$\mathsf{H}_1' = \mathsf{H}_2' +\!\!+\, x$	Lemma 6
$\mathsf{H}_1' \mid \mathsf{drop}\ x;\ e \longrightarrow_r \mathsf{H}_2' \mid e$	$(drop_r)$, $(dlam_r)$, $(dcon_r)$
$[\mathsf{H}_2]\lceil e \rceil = [\mathsf{H}_1]\lceil e \rceil = [\mathsf{H}_1']\lceil e \rceil = [\mathsf{H}_2']\lceil e \rceil$	follows

case $(dropru_h)$ $\mathsf{H} \mid r \leftarrow \mathsf{dropru}\ x;\ e \longrightarrow_h \mathsf{H}, z \mapsto^1 () \mid e[r{:=}z]$

$\mathsf{H}_1' \vdash r \leftarrow \mathsf{dropru}\ x;\ e \dashv \mathsf{H}_3$	given
$\mathsf{H}_1' \vdash r \leftarrow \mathsf{dropru}\ x;\ () \dashv \mathsf{H}_2'$	follows
$\mathsf{H}_2' \vdash e \dashv \mathsf{H}_3$	above
$\mathsf{fv}(e) \in \mathsf{H}_2'$	Lemma 5
$\mathsf{H}_1' +\!\!+\, r = \mathsf{H}_2' +\!\!+\, x$	Lemma 9
$\mathsf{H}_2'' = \mathsf{H}_2'[r{:=}z]$	let
$\mathsf{H}_1' \mid r \leftarrow \mathsf{dropru}\ x;\ e \longrightarrow_r \mathsf{H}_2'' \mid e[r{:=}z]$	$(drop_{ru})$, $(dlam_{ru})$, $(dcon_{ru})$
$[\mathsf{H}_2]\lceil e \rceil = [\mathsf{H}_1, z \mapsto^1 ()]\lceil e \rceil = [\mathsf{H}_1', z \mapsto^1 ()]\lceil e \rceil$	since z is fresh
$[\mathsf{H}_1', z \mapsto^1 ()]\lceil e \rceil = [\mathsf{H}_2'']\lceil e \rceil$	by LAM and DROPRU

$\square$

Lemma 18. (*Reference counting semantics is sound (big step)*)
If $\mathsf{H}_1 \mid \mathsf{E}[e_1] \longrightarrow_h \mathsf{H}_2 \mid \mathsf{E}[e_2]$, and $\mathsf{H}_1' \vdash \mathsf{E}[e_1] \dashv \mathsf{H}_3$, and $[\mathsf{H}_1]\lceil \mathsf{E}[e_1] \rceil = [\mathsf{H}_1']\lceil \mathsf{E}[e_1] \rceil$, then there exists H_2' such that $\mathsf{H}_1' \mid \mathsf{E}[e_1] \longrightarrow^*{}_r \mathsf{H}_2' \mid \mathsf{E}[e_2]$ and $[\mathsf{H}_2]\lceil \mathsf{E}[e_2] \rceil = [\mathsf{H}_2']\lceil \mathsf{E}[e_2] \rceil$.

Proof. (*Of Lemma 18*) By induction on E.

case $\mathsf{E} = \square$. Follows by Lemma 17.

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

$\mathsf{H}_1 \mid \mathsf{E}[e_1] \longrightarrow_h \mathsf{H}_2 \mid \mathsf{E}[e_2]$	given
$\mathsf{H}_1 \mid \mathsf{E}_1[e_1]\ x \longrightarrow_h \mathsf{H}_2 \mid \mathsf{E}_1[e_2]\ x$	given
$\mathsf{H}_1 \mid \mathsf{E}_1[e_1] \longrightarrow_h \mathsf{H}_2 \mid \mathsf{E}_1[e_2]$	follows
$\mathsf{H}_1' \vdash \mathsf{E}[e_1] \dashv \mathsf{H}_3$	given
$\mathsf{H}_1' \vdash e_1 \dashv \mathsf{H}_4$	follows
$[\mathsf{H}_1]\lceil \mathsf{E}[e_1] \rceil = [\mathsf{H}_1']\lceil \mathsf{E}[e_1] \rceil$	given
$[\mathsf{H}_1]\lceil \mathsf{E}_1[e_1] \rceil = [\mathsf{H}_1']\lceil \mathsf{E}_1[e_1] \rceil$	given
$\mathsf{H}_1' \mid \mathsf{E}_1[e_1] \longrightarrow^*{}_r \mathsf{H}_2' \mid \mathsf{E}_1[e_2]$	I.H.
$\mathsf{H}_1' \mid \mathsf{E}_1[e_1]\ x \longrightarrow^*{}_r \mathsf{H}_2' \mid \mathsf{E}_1[e_2]\ x$	follows
$\mathsf{H}_1' \mid \mathsf{E}[e_1] \longrightarrow^*{}_r \mathsf{H}_2' \mid \mathsf{E}[e_2]$	follows
$[\mathsf{H}_2]\lceil \mathsf{E}_1[e_1] \rceil = [\mathsf{H}_2']\lceil \mathsf{E}_1[e_1] \rceil$	above
$[\mathsf{H}_2]\lceil \mathsf{E}[e_2] \rceil = [\mathsf{H}_2']\lceil \mathsf{E}[e_2] \rceil$	follows

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

$\mathsf{H}_1 \mid \mathsf{E}[e_1] \longrightarrow_h \mathsf{H}_2 \mid \mathsf{E}[e_2]$	given
$\mathsf{H}_1 \mid \mathsf{let}\ x = \mathsf{E}_1[e_1]\ \mathsf{in}\ e \longrightarrow_h \mathsf{H}_2 \mid \mathsf{let}\ x = \mathsf{E}_1[e_2]\ \mathsf{in}\ e$	given
$\mathsf{H}_1 \mid \mathsf{E}_1[e_1] \longrightarrow_h \mathsf{H}_2 \mid \mathsf{E}_1[e_2]$	follows
$\mathsf{H}_1' \vdash \mathsf{E}[e_1] \dashv \mathsf{H}_3$	given
$\mathsf{H}_1' \vdash e_1 \dashv \mathsf{H}_4$	follows
$[\mathsf{H}_1]\lceil \mathsf{E}[e_1]\rceil = [\mathsf{H}_1']\lceil \mathsf{E}[e_1]\rceil$	given
$[\mathsf{H}_1]\lceil \mathsf{E}_1[e_1]\rceil = [\mathsf{H}_1']\lceil \mathsf{E}_1[e_1]\rceil$	given
$\mathsf{H}_1' \mid \mathsf{E}_1[e_1] \longrightarrow^*{}_r\ \mathsf{H}_2' \mid \mathsf{E}_1[e_2]$	I.H.
$\mathsf{H}_1' \mid \mathsf{let}\ x = \mathsf{E}_1[e_1]\ \mathsf{in}\ e \longrightarrow^*{}_r\ \mathsf{H}_2' \mid \mathsf{let}\ x = \mathsf{E}_1[e_2]\ \mathsf{in}\ e$	follows
$\mathsf{H}_1' \mid \mathsf{E}[e_1] \longrightarrow^*{}_r\ \mathsf{H}_2' \mid \mathsf{E}[e_2]$	follows
$[\mathsf{H}_2]\lceil \mathsf{E}_1[e_1]\rceil = [\mathsf{H}_2']\lceil \mathsf{E}_1[e_1]\rceil$	above
$[\mathsf{H}_2]\lceil \mathsf{E}[e_2]\rceil = [\mathsf{H}_2']\lceil \mathsf{E}[e_2]\rceil$	follows

□

Lemma 19. (*Reference counting semantics is sound (big step, star)*)
If $\mathsf{H}_1 \mid e_1 \longmapsto^*{}_h\ \mathsf{H}_2 \mid e_2$, and $\mathsf{H}_1' \vdash e_1 \dashv \mathsf{H}_3$, and $[\mathsf{H}_1]\lceil e_1\rceil = [\mathsf{H}_1']\lceil e_1\rceil$,
then there exists H_2' such that $\mathsf{H}_1' \mid e_1 \longmapsto^*{}_r\ \mathsf{H}_2' \mid e_2$ and $[\mathsf{H}_2]\lceil e_2\rceil = [\mathsf{H}_2']\lceil e_2\rceil$.

Proof. (*Of Lemma 19*) We can apply lemma 18 repeatedly: After each application, we know by lemma 16 that $\mathsf{H}_2' \vdash e_2 \dashv \mathsf{H}_3$.

Proof. (*Of Theorem 1*)

$\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$	given
$e \longmapsto^* v$	given
$e' \longmapsto^* v$	Theorem 6
$\varnothing \mid e' \longmapsto^*{}_h\ \mathsf{H}_2 \mid x$	Theorem 4
$[\mathsf{H}_2]x = v$	above
$\varnothing\ \mathsf{ok}$	
$e'\ \mathsf{ok}$	from LAM
$\varnothing \vdash e' \dashv \varnothing$	Lemma 12
$\varnothing \mid e' \longmapsto^*{}_r\ \mathsf{H}_3 \mid x$	Lemma 19
$[\mathsf{H}_3]x = [\mathsf{H}_2]x = v$	above

□

D.5 No Garbage

Theorem 7. (*No garbage*)
Given $\varnothing \mid \varnothing \vdash e \rightsquigarrow e_1$, 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 7*)

$\varnothing\ \mathsf{ok}$	by construction
$e_1\ \mathsf{ok}$	by LAM
$\varnothing \vdash e_1 \dashv \varnothing$	Lemma 12
$\mathsf{H}_i \vdash e_i \dashv \varnothing$	Lemma 16

□

Lemma 20. (*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 20*) By induction on the judgment.
case

$\mathsf{H}_0 \vdash C\ x_1 \ldots x_n \ \dashv \ \mathsf{H}_n$	given
$\mathsf{H}_0 \vdash x_1 \ \dashv \ \mathsf{H}_1 \ldots \mathsf{H}_{n-1} \vdash x_n \ \dashv \ \mathsf{H}_n$	DRCON
$\mathsf{reach}(\mathsf{H}_{i-1} \ - \ \mathsf{H}_i,\ \mathsf{H}_{i-1} \mid x_i)$	I.H.
$\mathsf{reach}(\mathsf{H}_{i-1} \ - \ \mathsf{H}_i,\ \mathsf{H}_0 \mid x_i)$	Lemma 5
$\mathsf{reach}(\mathsf{H}_n \ - \ \mathsf{H}_0,\ \mathsf{H}_0 \mid C\ x_1 \ldots x_n)$	Follows

case

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

case

$\mathsf{H},\ x \mapsto^{1} \lambda_{\overline{y}} z.\ e \vdash x \ \dashv \ \mathsf{H}_1$	given
$\mathsf{H} \vdash \overline{y} \ \dashv \ \mathsf{H}_1$	DRVARLAM
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \overline{y})$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lambda_{\overline{y}} z.\ e)$	by definition
$\mathsf{reach}((\mathsf{H}, x \mapsto^{1} \lambda_{\overline{y}} z.\ e) - \mathsf{H}_1,\ (\mathsf{H},\ x \mapsto^{1} \lambda_{\overline{y}} z.\ e) \mid x)$	follows

case

$\mathsf{H},\ x \mapsto^{1} C\ \overline{y} \vdash x \ \dashv \ \mathsf{H}_1$	given
$\mathsf{H} \vdash \overline{y} \ \dashv \ \mathsf{H}_1$	DRVARCON
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \overline{y})$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid C\ \overline{y})$	by definition
$\mathsf{reach}((\mathsf{H}, x \mapsto^{1} C\ \overline{y}) - \mathsf{H}_1,\ (\mathsf{H}, x \mapsto^{1} C\ \overline{y}) \mid x)$	follows

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

case

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

case

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

case

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

case

$\mathsf{H},\ x \mapsto^1 C\ \overline{y} \vdash r \leftarrow \mathsf{dropru}\ x;\ e\ \dashv \mathsf{H}_1$	given
$\mathsf{H},\ r \mapsto^1 () \vdash \mathsf{drop}\ \overline{y};\ e\ \dashv \mathsf{H}_1$	DRDROPCONRU
$\mathsf{reach}((\mathsf{H}, r \mapsto^1 ()) - \mathsf{H}_1,\ \mathsf{H} \mid \mathsf{drop}\ \overline{y};\ e)$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \mathsf{drop}\ \overline{y};\ e)$	follows
$\mathsf{reach}((\mathsf{H},\ x \mapsto^1 C\ \overline{y})\ -\ \mathsf{H}_1,\ (\mathsf{H},\ x \mapsto^1 C\ \overline{y}) \mid r \leftarrow \mathsf{dropru}\ x;\ e)$	follows

case

$\mathsf{H},\ x \mapsto^1 \lambda_{\overline{y}} z.\ e \vdash r \leftarrow \mathsf{dropru}\ x;\ e\ \dashv \mathsf{H}_1$	given
$\mathsf{H},\ r \mapsto^1 () \vdash \mathsf{drop}\ \overline{y};\ e\ \dashv \mathsf{H}_1$	DRDROPLAMRU
$\mathsf{reach}((\mathsf{H}, r \mapsto^1 ()) - \mathsf{H}_1,\ \mathsf{H} \mid \mathsf{drop}\ \overline{y};\ e)$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \mathsf{drop}\ \overline{y};\ e)$	follows
$\mathsf{reach}((\mathsf{H},\ x \mapsto^1 \lambda_{\overline{y}} z.\ e) - \mathsf{H}_1,\ (\mathsf{H},\ x \mapsto^1 \lambda_{\overline{y}} z.\ e) \mid r \leftarrow \mathsf{dropru}\ x;\ e)$	follows

case

$\mathsf{H} \vdash \lambda_{\overline{y}}\ x.\ e\ \dashv \mathsf{H}_1$	given
$\mathsf{H} \Vdash \overline{y}\ \dashv \mathsf{H}_1$	DRLAM
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \overline{y})$	I.H.
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lambda_{\overline{y}} x.\ e)$	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 5
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_2,\ \mathsf{H} \mid e_1\ e_2)$	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 5
$\mathsf{dom}(\mathsf{H}_1)\ \subseteq \mathsf{dom}(\mathsf{H}_1, x \mapsto^1 ())$	
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ (\mathsf{H}, x \mapsto^1 ())\ \mid e_2)$	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

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

□

Proof. (*Of Theorem 2*)

$\varnothing \mid \varnothing \vdash e\ \rightsquigarrow\ e'$	given
$\mathsf{H}_i \vdash e_i\ \dashv\ \varnothing$	Theorem 7
$\mathsf{reach}(\mathsf{H}_i\ -\ \varnothing,\ \mathsf{H}_i \mid e_i)$	Lemma 20

□

D.6 Precision

In this section we will define $(\star)\ =\ \mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ (\mathsf{H}_1, x \mapsto^1 ()) \mid \lceil e_2 \rceil)$.

Lemma 21. (*linear resource calculus relates to reference counting (Garbage free version)*)
If $\Delta \mid \Gamma \vdash_{\mathsf{GF}}\ e\ \rightsquigarrow\ e'$, then $[\![\Delta, \Gamma]\!] \vdash e'\ \dashv\ [\![\Delta]\!]$ (with $(\star)$ above).

Proof. (*Of Lemma 21*) Except for the LET case all other parts of the proof of lemma 12 remain unchanged.

$\forall y \in \Gamma_2.\ y \in \mathsf{fv}(e_2)$	(1), given
$\mathsf{reach}(\mathsf{fv}(e_2),\ [\![\Gamma_2, x]\!] \mid e_2)$	(2), by the definition of `reach`
$\mathsf{reach}(\Gamma_2,\ [\![\Gamma_2, x]\!] \mid e_2)$	(3), by (1) and (2)
$\mathsf{reach}([\![\Gamma_2, \Delta]\!]\ -\ [\![\Delta]\!],\ [\![\Gamma_2, x]\!] \mid e_2)$	(4), by (3)
$\mathsf{reach}([\![\Gamma_2, \Delta]\!] - [\![\Delta]\!],\ [\![\Gamma_2, x]\!] \mid \lceil e_2' \rceil)$	(5), by lemma 1

Lemma 22. (*Garbage-free at variables*)
If $\mathsf{H}_1 \vdash x\ \dashv\ \mathsf{H}_2$, then $\mathsf{reach}(\mathsf{H}_1\ -\ \mathsf{H}_2,\ \mathsf{H}_1 \mid \lceil x \rceil)$.

Proof. (*Of Lemma 22*) Follows from lemma 20, since $\lceil x \rceil\ =\ x$.

Lemma 23. (*Garbage-free*)
If $\mathsf{H}_1 \vdash \mathsf{E}[v] \dashv \mathsf{H}_2$, then $\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H}_1 \mid \lceil \mathsf{E}[v] \rceil)$.

Proof. (*Of Lemma 23*) By induction on the evaluation context.

case $\mathsf{E} = \Box$.

subcase $v = C\ \overline{x}$.

$\mathsf{H}_0 \vdash C\ x_1 \ldots x_n \dashv \mathsf{H}_n$	given
$\mathsf{H}_0 \vdash x_1 \dashv \mathsf{H}_1 \ldots \mathsf{H}_{n-1} \vdash x_n \dashv \mathsf{H}_n$	DRCON
$\mathsf{reach}(\mathsf{H}_{i-1} - \mathsf{H}_i,\ \mathsf{H}_{i-1} \mid x_i)$	Lemma 22
$\mathsf{reach}(\mathsf{H}_{i-1} - \mathsf{H}_i,\ \mathsf{H}_0 \mid x_i)$	Lemma 5
$\mathsf{reach}(\mathsf{H}_n - \mathsf{H}_0,\ \mathsf{H}_0 \mid C\ x_1 \ldots x_n)$	Follows
$\mathsf{reach}(\mathsf{H}_n - \mathsf{H}_0,\ \mathsf{H}_0 \mid \lceil C\ x_1 \ldots x_n \rceil)$	Follows

subcase $v = x$

$\mathsf{H}_1 \vdash x \dashv \mathsf{H}_2$	given
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H}_1 \mid \lceil x \rceil)$	Lemma 22

subcase $v = \lambda_{\overline{y}}\ x.\ e$

$\mathsf{H} \vdash \lambda_{\overline{y}}\ x.\ e \dashv \mathsf{H}_1$	given
$\mathsf{H} \vdash \overline{y} \dashv \mathsf{H}_1$	DRLAM
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \overline{y})$	Lemma 22
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lambda_{\overline{y}} x.\ e)$	follows
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lambda_{\overline{y}} x.\ \lceil e \rceil)$	follows
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil \lambda_{\overline{y}} x.\ e \rceil)$	follows

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

$\mathsf{H} \vdash \mathsf{E}_1[v]\ x \dashv \mathsf{H}_2$	given
$\mathsf{H} \vdash \mathsf{E}_1[v] \dashv \mathsf{H}_1$	DRAPP
$\mathsf{H}_1 \vdash x \dashv \mathsf{H}_2$	DRAPP
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil \mathsf{E}_1[v] \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H}_1 \mid \lceil x \rceil)$	Lemma 22
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ \mathsf{H} \mid \lceil x \rceil)$	Lemma 5
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_2,\ \mathsf{H} \mid \lceil \mathsf{E}_1[v]\ x \rceil)$	follows

case $\mathsf{E} = \mathsf{val}\ x = \mathsf{E}_1[v];\ e_2$.

$\mathsf{H} \vdash \mathsf{val}\ x = \mathsf{E}_1[v];\ e_2 \dashv \mathsf{H}_2$	given
$\mathsf{H} \vdash \mathsf{E}_1[v] \dashv \mathsf{H}_1$	DRBIND
$\mathsf{H}_1, x \mapsto^1 () \vdash e_2 \dashv \mathsf{H}_2$	DRBIND
$x \notin \mathsf{H},\ \mathsf{H}_2$	DRBIND
$\mathsf{reach}(\mathsf{H} - \mathsf{H}_1,\ \mathsf{H} \mid \lceil \mathsf{E}_1[v] \rceil)$	I.H.
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ (\mathsf{H}_1, x \mapsto^1 ()) \mid \lceil e_2 \rceil)$	(*)
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2,\ (\mathsf{H}, x \mapsto^1 ()) \mid \lceil e_2 \rceil)$	Lemma 5
$x \notin \mathsf{H}$	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

□

Proof. (*Of Theorem 3*)

$\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$	given
$\mathsf{H}_i \vdash e_i \dashv \varnothing$	Theorem 7
$\mathsf{reach}(\mathsf{H}_i - \varnothing,\ \mathsf{H}_i \mid \lceil \mathsf{E}[v] \rceil)$	Lemma 23

□

D.7 Frame-limited

In this section we will define $(\star) = \exists \mathsf{H}_1^1, \mathsf{H}_1^2 = \mathsf{H}_1.\ \mathsf{reach}(\mathsf{H}_1^1 - \mathsf{H}_2,\ (\mathsf{H}_1, x \mapsto^1 ())\mid \lceil e_2 \rceil) \ \wedge |\mathsf{H}_1^2| \leqslant c$.

Lemma 24. (*linear resource calculus relates to reference counting (Frame-limited version)*)
If $\Delta \mid \Gamma \vdash_{\mathsf{FL}} e \rightsquigarrow e'$, then $[\![\Delta, \Gamma]\!] \vdash e' \dashv [\![\Delta]\!]$ (with $(\star)$ above).

Proof. (*Of Lemma 21*) Except for the LET case all other parts of the proof of lemma 12 remain unchanged.

$\Gamma_2 = \Gamma', \Gamma''$ where $\Gamma' \subseteq \mathsf{fv}(e_2)$ and $\mathsf{sizeof}(\Gamma'') \leqslant c$	(1), given
$\mathsf{H}^1 = [\![\Gamma', \Delta]\!]$	(2), let
$\mathsf{reach}(\mathsf{fv}(e_2),\ [\![\Gamma_2, x]\!] \mid e_2)$	(3), by the definition of `reach`
$\mathsf{reach}(\mathsf{H}^1 - [\![\Delta]\!],\ [\![\Gamma_2, x]\!] \mid e_2)$	(4), by (2) and (3)
$\mathsf{reach}(\mathsf{H}^1 - [\![\Delta]\!],\ [\![\Gamma_2, x]\!] \mid \lceil e_2' \rceil)$	(5), by lemma 1
$\mathsf{H}^2 = [\![\Gamma_2, \Delta]\!] - \mathsf{H}^1 = \Gamma''$	(6), let
$\lvert\mathsf{H}^2\rvert \leqslant c$	(7), by (1)

With this $(\star)$ rule it is not clear that evaluation preserves it: After all, we are substituting new values into an expression and the values may be big. But if the $\mathsf{sizeof}(y)$ predicate is chosen such that no bigger value can be bound to y (for example, with help of a type system), then substitution preserves $(\star)$.

Lemma 25. (*Substitution preserves dependencies (Frame-limited version)*)
If $y, z \notin \mathsf{H}_1, \mathsf{H}_2$ and $\mathsf{H}_1, [\![y]\!] \vdash e \dashv \mathsf{H}_2$ and $\mathsf{sizeof}(y) = \mathsf{sizeof}(z)$, then $\mathsf{H}_1 +\!\!\!+\, z \vdash e[y{:=}z] \dashv \mathsf{H}_2$.

Proof. (*Of Lemma 25*) Except for the LET case all other parts of the proof of lemma 14 remain unchanged.

$\mathsf{H}, [\![y]\!] \vdash \mathsf{val}\ x = e_1;\ e_2 \dashv \mathsf{H}_2$	given
$\exists \mathsf{H}_1^1, \mathsf{H}_1^2 = \mathsf{H}_1.\ \mathsf{reach}(\mathsf{H}_1^1 - \mathsf{H}_2,\ (\mathsf{H}_1, x \mapsto^1 ()) \mid \lceil e_2 \rceil) \ \wedge \lvert\mathsf{H}_1^2\rvert \leqslant c$	given
$y \in \mathsf{H}_1$	else
$\mathsf{H} +\!\!\!+\, z \vdash e_1[y{:=}z] \dashv (\mathsf{H}_1 - [\![y]\!]) +\!\!\!+\, z$	by substitution and Lemma 13
$(\mathsf{H}_1 - [\![y]\!], [\![x]\!]) +\!\!\!+\, z \vdash e_2[y{:=}z] \dashv \mathsf{H}_2$	I.H.
$\mathsf{reach}((\mathsf{H}_1^1 - [\![y]\!] +\!\!\!+\, z) - \mathsf{H}_2,\ (\mathsf{H}_1 - [\![y]\!] +\!\!\!+\, z, x \mapsto^1 ()) \mid \lceil e_2[y{:=}z] \rceil)$	follows (if $y \in \mathsf{H}_1^1$)
$\lvert\mathsf{H}_1^2 - [\![y]\!] +\!\!\!+\, z\rvert \leqslant c$	follows (if $y \in \mathsf{H}_1^2$)
$\mathsf{H} +\!\!\!+\, z \vdash (\mathsf{val}\ x = e_1;\ e_2)[y{:=}z] \dashv \mathsf{H}_2$	DRBIND

Lemma 26. (*Frame-limited*)
If $\mathsf{H}_1 \vdash \mathsf{E}[v] \dashv \mathsf{H}_2$, there are $\mathsf{H}_1^1, \mathsf{H}_1^2 = \mathsf{H}_1$ such that $\mathsf{reach}(\mathsf{H}_1^1 - \mathsf{H}_2,\ \mathsf{H}_1 \mid \lceil \mathsf{E}[v] \rceil)$ and $\lvert\mathsf{H}_1^2\rvert \leqslant c \cdot \mid \mathsf{E} \mid$

Proof. (*Of Lemma 23*) By induction on the evaluation context.
case $\mathsf{E} = \square$.

By lemma 23, we can set $\mathsf{H}_1^1 = \mathsf{H}_1,\ \mathsf{H}_1^2 = \varnothing$. **case** $\mathsf{E} = \mathsf{E}_1\ x$.

$\mathsf{H} \vdash \mathsf{E}_1[v]\ x\ \dashv \mathsf{H}_2$	given
$\mathsf{H} \vdash \mathsf{E}_1[v]\ \dashv \mathsf{H}_1$	DRAPP
$\mathsf{H}_1 \vdash x\ \dashv \mathsf{H}_2$	DRAPP
$\mathsf{reach}(\mathsf{H}'^1 - \mathsf{H}_1, \mathsf{H} \mid \lceil \mathsf{E}_1[v] \rceil)$	I.H.
$\lvert \mathsf{H}^2 \rvert \leqslant c \cdot \lvert \mathsf{E}_1 \rvert$	I.H.
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2, \mathsf{H}_1 \mid \lceil x \rceil)$	Lemma 22
$\mathsf{reach}(\mathsf{H}_1 - \mathsf{H}_2, \mathsf{H} \mid \lceil x \rceil)$	Lemma 5
$\mathsf{H}^1 = \mathsf{H}'^1, (\mathsf{H}_1 - \mathsf{H}_2)$	let
$\mathsf{reach}(\mathsf{H}^1 - \mathsf{H}_2, \mathsf{H} \mid \lceil \mathsf{E}_1[v]\ x \rceil)$	follows
$\lvert \mathsf{H}^2 \rvert \leqslant c \cdot \lvert \mathsf{E} \rvert$	follows

case $\mathsf{E} = \mathsf{val}\ x = \mathsf{E}_1[v];\ e_2$.

$\mathsf{H} \vdash \mathsf{val}\ x = \mathsf{E}_1[v];\ e_2\ \dashv \mathsf{H}_2$	given
$\mathsf{H} \vdash \mathsf{E}_1[v]\ \dashv \mathsf{H}_1$	DRBIND
$\mathsf{H}_1, x \mapsto^1 () \vdash e_2\ \dashv \mathsf{H}_2$	DRBIND
$x \notin \mathsf{H}, \mathsf{H}_2$	DRBIND
$\mathsf{reach}(\mathsf{H}^1 - \mathsf{H}_1, \mathsf{H} \mid \lceil \mathsf{E}_1[v] \rceil)$	I.H.
$\lvert \mathsf{H}^2 \rvert \leqslant c \cdot \lvert \mathsf{E}_1 \rvert$	I.H.
$\mathsf{reach}(\mathsf{H}^1_1 - \mathsf{H}_2, (\mathsf{H}_1, x \mapsto^1 ()) \mid \lceil e_2 \rceil)$	(*)
$\lvert \mathsf{H}^2_1 \rvert \leqslant c$	(*)
$\mathsf{reach}(\mathsf{H}^1_1 - \mathsf{H}_2, (\mathsf{H}, x \mapsto^1 ()) \mid \lceil e_2 \rceil)$	Lemma 5
$\mathsf{H}_3 = (\mathsf{H}^1 - \mathsf{H}_1), \mathsf{H}^1_1$	let
$\mathsf{H}_4 = \mathsf{H}^2, \mathsf{H}^2_1$	let
$x \notin \mathsf{H}$	known
$x \notin \mathsf{dom}(\mathsf{H}) - \mathsf{dom}(\mathsf{H}_2)$	follows
$\mathsf{reach}(\mathsf{H}_3 - \mathsf{H}_2, \mathsf{H} \mid \lceil \mathsf{val}\ x = e_1;\ e_2 \rceil)$	follows
$\lvert \mathsf{H}_4 \rvert \leqslant c \cdot (\lvert \mathsf{E}_1 \rvert + 1) = c \cdot \lvert \mathsf{E} \rvert$	follows

□

Proof. (*Of Theorem 4*)

$\varnothing \mid \varnothing \vdash e \rightsquigarrow e'$	given
$\mathsf{H}_i \vdash e_i\ \dashv \varnothing$	Theorem 7
$\mathsf{reach}(\mathsf{H}^1_i - \varnothing, \mathsf{H}_i \mid \lceil \mathsf{E}[v] \rceil)$	Lemma 26
$\lvert \mathsf{H}^2_i \rvert \leqslant c \cdot \lvert \mathsf{E} \rvert$	above
$\mathsf{H}_i = \mathsf{H}^1_i, \mathsf{H}^2_i$	above

□

D.8 Reuse analysis is frame-limited

Proof. (*Of Theorem 5*) We can prove this by induction on the reuse judgment $\mathsf{S} \mid \mathsf{R} \Vdash e' \rightsquigarrow e''$.

case RVAR.

$\mathsf{S} \mid \mathsf{R} \Vdash x \rightsquigarrow x$	(1), given
$\Delta \mid x \vdash_{\mathsf{GF}} x \rightsquigarrow x$	(2), given
$\Delta \mid x, \varnothing \vdash_{\mathsf{FL}} x \rightsquigarrow x$	(3), by VAR

case RCON.

$\mathsf{S} \mid \mathsf{R} \Vdash C\ \overline{x} \rightsquigarrow C\ \overline{x}$	(1), given
$\Delta \mid \overline{x} \vdash_{\mathsf{GF}} C\ \overline{x} \rightsquigarrow C\ \overline{x}$	(2), given
$\Delta \mid \overline{x}, \varnothing \vdash_{\mathsf{FL}} C\ \overline{x} \rightsquigarrow C\ \overline{x}$	(3), by CON.

case RLAM.

$e' \;=\; \lambda_\Gamma\, x.\, \underline{e'}$	(1), given
$e'' \;=\; \lambda_\Gamma\, x.\, \underline{e''}$	(2), given
$\Delta \mid \Gamma \vdash_{\mathsf{GF}} \lambda x.\, \underline{e} \rightsquigarrow \lambda_\Gamma\, x.\, \underline{e'}$	(3), given and by (1)
$\varnothing \mid \Gamma, x \vdash_{\mathsf{GF}} \underline{e} \rightsquigarrow \underline{e'}$	(4), by (3)
$\mathsf{S} \mid \varnothing \Vdash \underline{e'} \rightsquigarrow \underline{e''}$	(5), given
$\varnothing \mid \Gamma, x, \varnothing \vdash_{\mathsf{FL}} \underline{e} \rightsquigarrow \underline{e''}$	(6), by induction
$\Delta \mid \Gamma, \varnothing \vdash_{\mathsf{FL}} \lambda x.\, \underline{e} \rightsquigarrow \lambda_\Gamma\, x.\, \underline{e''}$	(7), by LAM

case RAPP.

$e' \;=\; \underline{e'}\; x$	(1), given
$e'' \;=\; \underline{e''}\; x$	(2), given
$\Delta \mid \Gamma \vdash_{\mathsf{GF}} \underline{e}\; x \rightsquigarrow \underline{e'}\; x$	(3), given and by (1)
$\Delta, x \mid \Gamma \vdash_{\mathsf{GF}} \underline{e} \rightsquigarrow \underline{e'}$	(4), by (3)
$\mathsf{S} \mid \mathsf{R} \Vdash \underline{e'} \rightsquigarrow \underline{e''}$	(5), given
$\Delta, x \mid \Gamma, \mathsf{R} \vdash_{\mathsf{FL}} \underline{e} \rightsquigarrow \underline{e''}$	(6), by induction
$\Delta \mid \Gamma, \mathsf{R} \vdash_{\mathsf{FL}} \underline{e}\; x \rightsquigarrow \underline{e''}\; x$	(7), by APP

case RDROP-REUSE.

$\Delta \mid \Gamma \vdash_{\mathsf{GF}} e \rightsquigarrow e'$	(1), given
$\mathsf{S} \mid \mathsf{R} \Vdash e' \rightsquigarrow e''$	(2), given
$\Delta \mid \Gamma, \mathsf{R} \vdash_{\mathsf{FL}} e \rightsquigarrow e''$	(3), by induction
$\Delta \mid \Gamma, \mathsf{R}, r \vdash_{\mathsf{FL}} e \rightsquigarrow \mathsf{drop}\ r;\ e''$	(4), by DROP

case RLET.

$\mathsf{S} \mid \mathsf{R}_1, \mathsf{R}_2 \Vdash \mathsf{let}\ x = e'_1\ \mathsf{in}\ e'_2 \rightsquigarrow \mathsf{let}\ x = e''_1\ \mathsf{in}\ e''_2$	(1), given
$\mathsf{S} \mid \mathsf{R}_1 \Vdash e'_1 \rightsquigarrow e''_1$	(2), by (1)
$\mathsf{S} \mid \mathsf{R}_2 \Vdash e'_2 \rightsquigarrow e''_2$	(3), by (1)
$\Delta \mid \Gamma_1, \Gamma_2 \vdash_{\mathsf{GF}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{let}\ x = e'_1\ \mathsf{in}\ e'_2$	(4), given
$\Delta, \Gamma_2 \mid \Gamma_1 \vdash_{\mathsf{GF}} e_1 \rightsquigarrow e'_1$	(5), by (4)
$\Delta \mid \Gamma_2, x \vdash_{\mathsf{GF}} e_2 \rightsquigarrow e'_2$	(6), by (4)
$(\star)_{gf}$ for Γ_2	(7), by (4)
$(\star)_{fl}$ for Γ_2	(8), by (7) and monotonicity
$\Delta, \Gamma_2 \mid \Gamma_1, \mathsf{R}_1 \vdash_{\mathsf{FL}} e_1 \rightsquigarrow e'_1$	(9), by induction on (5) and (2)
$\Delta \mid \Gamma_2, \mathsf{R}_2, x \vdash_{\mathsf{FL}} e_2 \rightsquigarrow e'_2$	(10), by induction on (6) and (3)
$(\star)_{fl}$ for R_2	(11), since reuse tokens are small
$\Delta \mid \Gamma_1, \Gamma_2, \mathsf{R} \vdash_{\mathsf{FL}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{let}\ x = e''_1\ \mathsf{in}\ e''_2$	(12), by LET and (9), (10), (8) and (11)

case RMATCH.

$\Delta \mid \Gamma, x \vdash_{\mathsf{GF}} \mathsf{match}\ x\ \{\overline{p_i \mapsto e_i}\} \rightsquigarrow \mathsf{match}\ x\ \{\overline{p_i \mapsto \mathsf{dup}(\overline{z_i});\ e'_i}\}$	(1), given
$\Delta \mid \Gamma, x, \overline{z_i} \vdash_{\mathsf{GF}} e_i \rightsquigarrow e'_i$	(2), by (1)
$\mathsf{S} \mid \mathsf{R} \Vdash \mathsf{match}\ x\ \{\overline{p_i \mapsto \mathsf{dup}(\overline{z_i});\ e'_i}\} \rightsquigarrow \mathsf{match}\ x\ \{\overline{p_i \mapsto e''_i}\}$	(3), given
$\mathsf{S},\ x : n \mid \mathsf{R} \Vdash \mathsf{dup}(\overline{z_i});\ e'_i \rightsquigarrow e''_i$	(4), by (3)
$\mathsf{S},\ x : n \mid \mathsf{R} \Vdash e'_i \rightsquigarrow \underline{e''_i}$	(5), by (3) and repeatedly invoking RDU
$\Delta \mid \Gamma, x, \overline{z_i}, \mathsf{R} \vdash_{\mathsf{FL}} e_i \rightsquigarrow \underline{e''_i}$	(6), by induction on (2) and (5)
$\Delta \mid \Gamma, x, \mathsf{R} \vdash_{\mathsf{FL}} \mathsf{match}\ x\ \{\overline{p_i \mapsto e_i}\} \rightsquigarrow \mathsf{match}\ x\ \{\overline{p_i \mapsto \mathsf{dup}(\overline{z_i});\ e''_i}\}$	(7), by MATCH

case RDUP.

$$
\begin{array}{ll}
\mathsf{S} \mid \mathsf{R} \Vdash \mathsf{dup}\ x;\ e' \rightsquigarrow \mathsf{dup}\ x;\ e'' & \text{(1), given} \\
\mathsf{S} \mid \mathsf{R} \Vdash e' \rightsquigarrow e'' & \text{(2), by (1)} \\
\Delta \mid \Gamma \vdash_{\mathsf{GF}} e \rightsquigarrow \mathsf{dup}\ x;\ e' & \text{(3), given} \\
\Delta \mid \Gamma, x \vdash_{\mathsf{GF}} e \rightsquigarrow e' & \text{(4), by (2)} \\
\Delta \mid \Gamma, x, \mathsf{R} \vdash_{\mathsf{FL}} e \rightsquigarrow e'' & \text{(5), by induction} \\
\Delta \mid \Gamma, \mathsf{R} \vdash_{\mathsf{FL}} e \rightsquigarrow \mathsf{dup}\ x;\ e'' & \text{(6), by DUP}
\end{array}
$$

case RDROP.

$$
\begin{array}{ll}
\mathsf{S} \mid \mathsf{R} \Vdash \mathsf{drop}\ x;\ e' \rightsquigarrow \mathsf{drop}\ x;\ e'' & \text{(1), given} \\
\mathsf{S} \mid \mathsf{R} \Vdash e' \rightsquigarrow e'' & \text{(2), by (1)} \\
\Delta \mid \Gamma, x \vdash_{\mathsf{GF}} e \rightsquigarrow \mathsf{drop}\ x;\ e' & \text{(3), given} \\
\Delta \mid \Gamma \vdash_{\mathsf{GF}} e \rightsquigarrow e' & \text{(4), by (2)} \\
\Delta \mid \Gamma, \mathsf{R} \vdash_{\mathsf{FL}} e \rightsquigarrow e'' & \text{(5), by induction} \\
\Delta \mid \Gamma, x, \mathsf{R} \vdash_{\mathsf{FL}} e \rightsquigarrow \mathsf{drop}\ x;\ e'' & \text{(6), by DROP}
\end{array}
$$

case RDROPRU.

$$
\begin{array}{ll}
\mathsf{S} \mid \mathsf{R} \Vdash r \leftarrow \mathsf{dropru}\ x;\ e' \rightsquigarrow r \leftarrow \mathsf{dropru}\ x;\ e'' & \text{(1), given} \\
\mathsf{S} \mid \mathsf{R}, r \Vdash e' \rightsquigarrow e'' & \text{(2), by (1)} \\
\Delta \mid \Gamma, x \vdash_{\mathsf{GF}} e \rightsquigarrow r \leftarrow \mathsf{dropru}\ x;\ e' & \text{(3), given} \\
\Delta \mid \Gamma, r \vdash_{\mathsf{GF}} e \rightsquigarrow e' & \text{(4), by (2)} \\
\Delta \mid \Gamma, \mathsf{R}, r \vdash_{\mathsf{FL}} e \rightsquigarrow e'' & \text{(5), by induction} \\
\Delta \mid \Gamma, \mathsf{R}, r \vdash_{\mathsf{FL}} e \rightsquigarrow r \leftarrow \mathsf{dropru}\ x;\ e'' & \text{(6), by DROPRU}
\end{array}
$$

case RREUSE-DROP.

$$
\begin{array}{ll}
\mathsf{S} \mid \mathsf{R} \Vdash \mathsf{drop}\ x;\ e' \rightsquigarrow r \leftarrow \mathsf{dropru}\ x;\ e'' & \text{(1), given} \\
\mathsf{S} \mid \mathsf{R}, r : n \Vdash e' \rightsquigarrow e'' & \text{(2), by (1)} \\
\Delta \mid \Gamma, x \vdash_{\mathsf{GF}} e \rightsquigarrow \mathsf{drop}\ x;\ e' & \text{(3), given} \\
\Delta \mid \Gamma \vdash_{\mathsf{GF}} e \rightsquigarrow e' & \text{(4), by (2)} \\
\Delta \mid \Gamma, \mathsf{R}, r \vdash_{\mathsf{FL}} e \rightsquigarrow e'' & \text{(5), by induction} \\
\Delta \mid \Gamma, x, \mathsf{R} \vdash_{\mathsf{FL}} e \rightsquigarrow r \leftarrow \mathsf{dropru}\ x;\ e'' & \text{(6), by DROPRU}
\end{array}
$$