FP^2: Fully in-Place Functional Programming

ANTON LORENZEN, University of Edinburgh, UK
DAAN LEIJEN, Microsoft Research, USA
WOUTER SWIERSTRA, Universiteit Utrecht, Netherlands

As functional programmers we always face a dilemma: should we write purely functional code, or sacrifice purity for efficiency and resort to in-place updates? This paper identifies precisely when we can have the best of both worlds: a wide class of purely functional programs can be executed safely using in-place updates without requiring allocation, provided their arguments are not shared elsewhere.

We describe a linear *fully in-place* (FIP) calculus where we prove that we can always execute such functions in a way that requires no (de)allocation and uses constant stack space. Of course, such a calculus is only relevant if we can express interesting algorithms; we provide numerous examples of in-place functions on datastructures such as splay trees or finger trees, together with in-place versions of merge sort and quick sort.

We also show how we can generically derive a map function over *any* polynomial data type that is fully in-place. Finally, we have implemented the rules of the FIP calculus in the Koka language. Using the Perceus reference counting garbage collection, this implementation dynamically executes FIP functions in-place whenever possible.

CCS Concepts: • **Software and its engineering** → **Control structures**; *Recursion*; • **Theory of computation** → **Operational semantics**.

Additional Key Words and Phrases: FBIP, Tail Recursion Modulo Cons

ACM Reference Format:
Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 198 (August 2023), 30 pages. https://doi.org/10.1145/3607840

1 INTRODUCTION AND OVERVIEW

The functional program for reversing a list in linear time using an accumulating parameter has been known for decades, dating back at least as far as Hughes's work on difference lists [1986]:

```
fun reverse-acc( xs : list<a>, acc : list<a> ) : list<a>
  match xs
    Cons(x,xx) -> reverse-acc( xx, Cons(x,acc) )
    Nil        -> acc

fun reverse( xs : list<a> ) : list<a>
  reverse-acc(xs,Nil)
```

As this definition is *pure*, we can calculate with it using equational reasoning in the style of Bird and Meertens [Backhouse 1988; Gibbons 1994]. Using simple induction, we can, for instance, prove that this linear time list reversal produces the same results as its naive quadratic counterpart.

Not all in the garden is rosy: what about the function's memory usage? The purely functional definition of `reverse` allocates fresh `Cons` nodes in each iteration; an automatic garbage collector needs to discard unused memory. This generally induces a performance penalty relative to an imperative

Authors' addresses: Anton Lorenzen, University of Edinburgh, School of Informatics, Edinburgh, UK, anton.lorenzen@ed.ac.uk; Daan Leijen, Microsoft Research, Redmond, WA, USA, daan@microsoft.com; Wouter Swierstra, Universiteit Utrecht, Utrecht, Netherlands, w.s.swierstra@uu.nl.

2475-1421/2023/8-ART198
https://doi.org/10.1145/3607840

in-place implementation that destructively updates the pointers of a linked list. Reasoning about such imperative in-place algorithms, however, is much more difficult.

As programmers we seem to face a dilemma: should we write purely functional code, or sacrifice purity for efficiency and resort to in-place updates? This paper identifies precisely when we can have the best of both worlds: a wide class of purely functional programs can be executed safely using in-place updates without requiring allocation, including the `reverse` function above.

In particular, what *if* the compiler can make the assumption that the function parameters are *owned* and unique at runtime, i.e. that there are no other references to the input list `xs` of `reverse` at runtime. In that case, the compiler can safely *reuse* any matched `Cons` node and update it in-place with the result – effectively updating the list in-place. In this paper we describe a novel *fully in-place* (FIP) calculus that guarantees that such a function can be compiled in a way to never (de)allocate memory or use unbounded stack space—it can be executed fully in-place.

To illustrate the purely functional fully in-place paradigm, we consider *splay trees* as described by Sleator and Tarjan [1985]. These are self-balancing trees where every access to an element in the tree, including lookup, restructures the tree such that the element is "splayed" to the top of the tree. As a result, the `lookup` function not only returns a boolean representing whether or not the element was found in the tree, but also a newly splayed tree. Splay trees are generally not considered well-suited for functional languages, because every such restructuring of the tree copies the spine of the tree leading to decreased performance relative to an imperative implementation that can rebalance the tree in-place. Surprisingly, it turns out to be possible to write the splay algorithms in a purely functional style using *fully in-place* functions.

1.1 Zippers and Unboxed Tuples

Let us first define the type of splay trees containing integers[1]:

```
type stree
  Node(left : stree, value : int, right : stree)
  Leaf
```

For the `lookup` function, once we find a given element in the tree, we need to somehow navigate back through the tree to splay the node to the top. The usual imperative approach uses parent pointers for this, but in a purely functional style we can use Huet's *zipper* [1997] instead. The central idea is a simple one: to navigate through a tree step by step, we store the current subtree in focus together with its context. Naively, one might try to represent the context as a path from the root of the tree to the current subtree. The zipper, however, *reverses* this path so each step up or down the tree requires only constant time. For splay trees, we can define the corresponding zipper as:

```
type szipper
  Root
  NodeL( up : szipper, value : int, right : stree )
  NodeR( left : stree,  value : int, up : szipper )
```

Huet already observed that the zipper operations could be implemented in-place. The original paper presenting the zipper concludes with the following paragraph [Huet 1997]:

> Efficient destructive algorithms on binary trees may be programmed with these completely applicative primitives, which all use constant time, since they all reduce to local pointer manipulation.

Our FIP calculus provides the language to make such a statement precise: using the rules presented in the next section we can check statically that the various operations on zippers are indeed fully

[1] All examples in this paper are written in the Koka langugage which has a full implementation of the FIP check (v2.4.2).

in-place. For example, we can move focus to the left subtree as follows:

```
fip fun left(t : stree, ctx : szipper) : (stree,szipper)
  match! t
    Node(l,x,r) -> (l, NodeL(ctx,x,r))
    Leaf        -> (Leaf, ctx)
```

The `fip` keyword indicates that a static check guarantees the function is fully in-place. This check, specified in Section 2, verifies that the function lives in a *linear* fragment of the language where the function parameters and variables are *owned* and can only be used once. As a consequence, we can safely reuse the memory of a linear value in-place once it is no longer used. The `match!` keyword[2] indicates a *destructive match*, after which the matched variable `t` can no longer be used. Intuitively, we can then see straightaway that the matched `Node` cell has become redundant and can be reused for the `NodeL` cell of the zipper.

To make this intuition more precise, we view a destructively matched constructor, such as `Node(l,x,r)`, as a collection of its children `l`, `x`, and `r`, and a *reuse credit* $\diamond_3$. This "diamond" resource type $\diamond_k$ represents a specific heap cell of size k and is inspired by the work of Aspinall and Hofmann on space credits [Aspinall and Hofmann 2002; Aspinall et al. 2008; Hofmann 2000b 2000a]. Similar to their space credits, we also apply the linearity restriction to reuse credits, but, unlike space credits, a reuse credit can not be split or merged with other credits. In our example, the `match!` introduces a reuse credit $\diamond_3$ for the `Node`, which is consumed by the allocation of `NodeL` which allows our `left` function to be fully in-place.

It may seem that we still need to allocate a tuple to store the result. Such allocation, however, is usually unnecessary since tuples are often created only to hold multiple return values and immediately destructed afterwards. In our calculus and implementation, we model tuples as *unboxed* values hence no allocation is needed for these[3].

We can now also see that the list reversal of the introduction is also fully in-place:

```
fip fun reverse-acc( xs : list<a>, acc : list<a> ) : list<a>
  match! xs
    Cons(x,xx) -> reverse-acc( xx, Cons(x,acc) )
    Nil        -> acc
```

where the destructive match on `xs` allows the matched `Cons` cell to be reused ($\diamond_2$) for the `Cons(x,acc)` allocation in the branch.

1.2 Splay Tree Lookup and Atoms

Using the zipper definition for splay trees, we can now define the `lookup` function as follows:

```
fip fun lookup( t : stree, x : int ) : (bool, stree)
  zlookup(t,x,Root)

fip fun zlookup( t : stree, x : int, ctx : szipper ) : (bool, stree)
  match! t
    Leaf -> (False, splay-leaf(ctx))                // not found, but splay anyway
    Node(l,y,r) ->
      if   x < y then zlookup(l,x,NodeL(ctx,y,r))   // go down the left  (NodeL reuses Node)
      elif x > y then zlookup(r,x,NodeR(l,y,ctx))   // go down the right (NodeR reuses Node)
      else (True, splay(Top(l,y,r),ctx))            // found it, now splay it to the top
```

The `lookup` function calls `zlookup` with an initial empty context `Root`. It seems we need to allocate the `Root` constructor, but constructors without any fields do not require allocation. These are

[2] In our implementation it turns out we can always *infer* when a match needs to be destructive and we can always write just `match` for both destructive and borrowing matches. However, for clarity and correspondence to our formal FIP calculus, we denote destructive matches explicitly in this paper.

[3] In Koka, tuples are implemented as *value types* which are unboxed and passed in registers. The `fip` keyword additionally checks that no automatic (heap allocated) boxing is applied for such value types inside a FIP function.

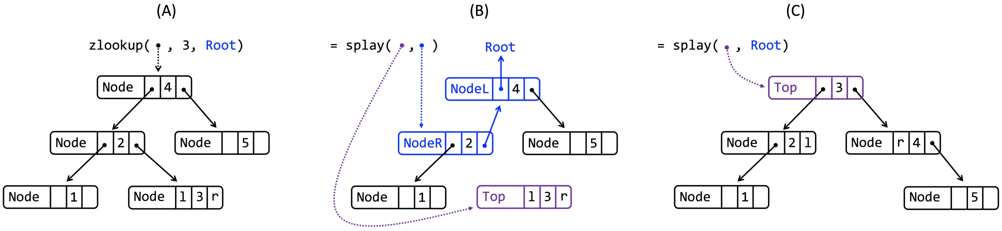

Fig. 1. Looking up the number 3 in a splay tree (A), creating the zipper context to the node containing 3 (B), and splaying the node up to the top (C).

typically implemented using pointer-tagging where only the tag is used to represent them. We call such constructors *atoms*. Examples include the `Nil` of lists, but also booleans (`True` and `False`), and, depending on the implementation, primitive types like integers (`int`) or floats.

The `zlookup` function traverses down the tree while extending the current zipper context *in-place* with the path that is followed. Figure 1 shows a concrete example of how `zlookup` constructs the zipper in the transition from (A) to (B). Once the element is found, the corresponding node is splayed back up to the top of the tree as `splay(Top(l,x,r),ctx)`. Here we use the `Top` constructor:

```
type top
  Top( left : stree, value : int, right : stree )
```

But why is this `Top` constructor needed? Can we not just call `splay` directly with explicit arguments as `splay(l,x,r,ctx)`? This is not possible though in a fully in-place way. In particular, it would mean that the `Node(l,y,r)` on which we matched would need to be deallocated as it cannot be reused immediately (and deallocation is not allowed in `fip` functions). Moreover, we would now need to allocate the final top node later on. If we define `splay` without using `Top`, we would get something like:

```
fun splay( l : stree, x : int, r : stree, ctx : szipper ) : stree  // not fip!
  match! ctx
    Root -> Node(l,x,r)
    ...
```

That is, once the zipper `ctx` is at the `Root` we need to return a fresh `Node` with `x` on top. As there is no constructor that can be reused (since `Root` is just an atom), this would require allocation. By using the intermediate `Top` constructor we avoid this: at the call site we can now reuse the `Node` that we just matched for `Top`, and later on when we reach the `Root`, we can reuse `Top` again to create the final `Node` that becomes the top of the returned splay tree. This results in the final fully in-place definition of the `splay` function:

```
fip fun splay( top : top , ctx : szipper ) : stree
  match! top
    Top(l,x,r) -> match! ctx
      Root                        -> Node(l,x,r)
      NodeL(Root,y,ry)            -> Node(l,x,Node(r,y,ry))                        // zig
      NodeL(NodeR(lz,z,up),y,ry)  -> splay( Top(Node(lz,z,l),x,Node(r,y,ry)), up)  // zig-zag
      NodeL(NodeL(up,z,rz),y,ry)  -> splay( Top(l,x,Node(r,y,Node(ry,z,rz))), up)  // zig-zig
      NodeR(ly,y,Root)            -> Node(Node(ly,y,l),x,r)
      NodeR(ly,y,NodeL(up,z,rz))  -> splay( Top(Node(ly,y,l),x,Node(r,z,rz)), up)  // (B)->(C)
      NodeR(ly,y,NodeR(lz,z,up))  -> splay( Top(Node(Node(lz,z,ly),y,l),x,r), up)
```

The compiler statically checks that this function is fully in-place. As a result, we know that its execution uses no stack space and performs all its rebalancing operations without any (de)allocation – each `Top` and `Node` can reuse a destructively matched `Top`, `NodeL`, or `NodeR` in every branch. Each of the matched cases correspond to a "zig", "zig-zig", and "zig-zag" rebalance operation as described by Sleator and Tarjan [1985]. Figure 1 shows a concrete example of the "zig-zag" in the transition from (B) to (C). For completeness, the auxiliary splay-leaf function that is called in case the item is not a member of the tree is included below:

```
fip fun splay-leaf( ctx : szipper ) : stree
  match! ctx
    Root          -> Leaf
    NodeL(up,x,r) -> splay(Top(Leaf,x,r),up)
    NodeR(l,x,up) -> splay(Top(l,x,Leaf),up)
```

The definition of splay may look somewhat involved but compared to the imperative definition it is fairly concise. Moreover, the usual imperative algorithm uses extra space for parent pointers in each node. We do not need this: if we study the generated code for the fip functions, we see that the reuse of the zipper nodes corresponds to the usual "pointer-reversal" techniques [Schorr and Waite 1967] (which is illustrated nicely in Figure 1 (B)). Such pointer-reversal is not often used explicitly in practice though since it is difficult to get right by hand. In the code above, however, the fully in-place fip functions using zippers provide a statically typed, memory safe, purely functional definition with the same runtime behaviour, without requiring explicit pointer manipulation.

1.3 Borrowing and Second-Order Functions

While our examples have so far been entirely first-order, we also allow functions to be passed as arguments. For example, we can map over a splay tree as follows:

```
fbip fun smap( t : stree, ^f : int -> int) : stree
  match! t
    Node(l,v,r) -> Node(smap(f,l), f(v), smap(f,r))
    Leaf        -> Leaf
```

In this function we seemingly violate our linearity constraint since f is used three times in the first branch. However, the function parameter is marked as *borrowed* using the hat notation (^f) [Ullrich and de Moura 2019]. This allows the parameter to be freely shared but at the same time it cannot be used in a destructive match, passed as an owned parameter, or returned as a result. Such borrowing is often useful for functions that inspect a data structure. Consider the following example is-node function:

```
fip is-node( ^t : stree ) : bool
  match t
    Node(_,_,_) -> True
    _           -> False
```

This function would not be fully in-place if we matched destructively. A subtle point about higher-order functions is that we consider an application f(e) as a borrowed use of f, and, as a consequence, f cannot modify any captured free variables in-place. We enforce this in our calculus by only allowing top-level functions (rather than arbitrary closures) as arguments in our fully in-place calculus, effectively making it second-order.

Finally, note that the smap function is marked not as fip but as fbip. Unlike the earlier splay function, smap has recursive calls in non-tail positions. That makes it hard to claim that this function is "fully in-place"— after all, its execution uses stack space linear in the depth of the splay tree. We use the fbip keyword to signify FIP functions that still reuse in-place but are allowed to use arbitrary stack space and deallocate memory. Nevertheless, for smap this is not really required: in Section 3 we show that *any* map function of polynomial datatypes (including smap) can be tranformed into a tail-recursive, zipper-based traversal that *is* fully in-place.

The fbip keyword is derived from the "functional but in-place" technique [Reinking, Xie et al. 2021] which is a more liberal notion of our strict fully in-place functions. Our implementation also supports fip(n) and fbip(n) for a constant n, which allows the function to allocate at most a constant n constructors. This is sometimes useful for functions like splay tree insertion where a single `Node` may need to be allocated for the newly inserted element, making it `fip(1)`.

1.4 Fully in-Place in a Functional World

One might argue that fully in-place programming is just imperative programming in functional clothing. Where are the closures, the non-linear values, the persistence? And who allocates the list to be reversed in the first place? We need to be able to embed our fully in-place functions in a larger host language to be useful. The challenge is to do this safely while still guaranteeing in-place updates when possible. To illustrate this point, consider the following function:

```
fun palindrome( xs : list<a>) : list<a>
  append(xs, reverse(xs))
```

Even though `reverse` is a `fip` function, it would not be safe for it to destructively update its input list since the argument `xs` is used twice (as an owned argument) in the body of `palindrome`. The FIP calculus presented here statically checks a function's definition—yet deciding which *calls* to `fip` functions can be safely executed using destructive updates requires further information about how arguments are shared at call sites.

1.4.1 Uniqueness Typing. One way to check this information statically is using a *uniqueness* type system. For example, Clean [Brus et al. 1987] is a functional language where the type system tracks when arguments are *unique* or *shared* [Barendsen and Smetsers 1995; De Vries et al. 2008]. A `fip` function may safely use in-place mutation, provided all owned parameters have a *unique type*. In that way, the type system guarantees that any argument passed at runtime will be a unique reference to that object, ruling out any possible sharing. As a result, it is always safe to reuse the argument in-place. One possible drawback of linear type systems and uniqueness typing is that it leads to code duplication, where a single function can have multiple different implementations: one version taking a unique argument; and one taking a shared argument. For example, we may need to define two `reverse` functions that have an equivalent implementation but only differ in the `fip` annotation and the uniqueness type of the input list – one uses copying and can be used persistently with a shared list, while the FIP variant updates the list in-place but requires the input list to be unique.

1.4.2 Precise Reference Counting. Checking call sites of `fip` functions need not happen statically. Instead, we could also use a *dynamic* approach where we check at runtime if a FIP function can be executed in-place [Lorenzen and Leijen 2022; Schulte and Grieskamp 1992; Ullrich and de Moura 2019]. This is the approach taken in our implementation in the Koka language, which uses Perceus precise reference counting [Reinking, Xie et al. 2021; Ullrich and de Moura 2019]. When an object has a reference count of one, it is safe to update it in-place. To illustrate how the compiled code looks in practice, consider the compilation of the fully in-place `reverse-acc` function. The destructive match on the input list now dynamically checks whether or not the list can be mutated in place and the generated code will look something like:

```
fip fun reverse-acc( xs : list<a>, acc : list<a> ) : list<a>
  match! xs
    Cons(x,xx) ->
      val ru = if is-unique(xs) then &xs else { dup(x); dup(xx); decref(xs); alloc(2) }
      reverse-acc( xx, Cons@ru(x,acc) )
    Nil -> acc
```

The reuse credit $\diamond_2$ is compiled into an explicitly named reuse token `ru`, and holds to the memory location of the resulting `Cons` cell. If the input list is unique, we *reuse* the address of the input, `&xs`, and otherwise, we adjust the reference counts of the children accordingly and return freshly allocated memory of the right size. In the recursive call, we initialize the `Cons` cell in-place at the `ru` memory location as `Cons@ru(x,acc)`. Compared to the static analysis, we have lost the static guarantee that the owned parameters are unique at runtime, but also we gained expressiveness: in particular, we can define now a *single* purely functional but fully in-place `reverse` function that serves all usage scenarios: it efficiently updates the elements in place if the argument list is unique at runtime, but it also adapts dynamically when the list, or any sublist happens to be shared – falling back gracefully to allocating fresh `Cons` nodes for the resulting list.

1.5 Contributions

To support the motivating examples outlined so far, this paper makes the following contributions:

- Following the pioneering work on the LFPL calculus [Hofmann 2000b 2000a], we present a novel *fully in-place* (FIP) calculus (Section 2), precisely capturing those functions that can be executed fully in-place. We provide a standard functional operational semantics for our language, but also define an equivalent semantics for FIP functions in terms of a fixed store, where no (de)allocation can take place. As a result, we know that FIP functions never allocate memory and use bounded stack space. As shown for splay trees, *atoms* and *unboxed tuples* are needed to avoid allocations for many common scenarios and the FIP calculus includes these features. Furthermore, the rules of the FIP calculus provide a static guarantee of linearity in a *syntactic* way where parameters can be *owned* uniquely or *borrowed*.
- The FIP calculus is only useful if it can actually be used to describe interesting algorithms. To show the wide applicability of our approach we present a variety of familiar functional programs and operations on datastructures that are all *fully in-place*. We have already seen how Huet's *zipper* [1997] datastructure, colloquially described as the functional equivalent of backpointers, can be used fully in-place, and how we can use this to implement fully in-place *splay trees* [Sleator and Tarjan 1985]. In Section 3, we further show that we can use a defunctionalized CPS transformation [Danvy 2008] to derive a generic map function for *any* polynomial inductive datatype. The derived map uses fully in-place Schorr-Waite traversal [Schorr and Waite 1967] without using any extra stack- or heap space. In Section 4, we show further examples of functional algorithms and datastructures that are fully in-place, including imperative red-black tree insertion [Cormen et al. 2022], `cons` and `append` operations on finger trees [Claessen 2020; Hinze and Paterson 2006], and even sorting algorithms like merge sort and quick sort. We have an implementation of the FIP calculus in a fork of the Koka compiler that can check and compile all examples in this paper.
- Finally, we study the dynamic embedding of our FIP calculus based on precise reference counting in detail in Section 5. Integrating the static FIP calculus with the dynamic Perceus linear resource calculus, λ^1 [Lorenzen and Leijen 2022; Reinking, Xie et al. 2021], gives us precisely the information we need to decide when a function call can be executed in-place or not. However, the original linear resource calculus does not model reuse, atoms, unboxing, or borrowing, all of which are essential for FIP programs. We simplify and extend the linear resource calculus into a new calculus (λ^{fip}) which includes all these features, and give a novel proof of soundness.
  Surprisingly, it turns out that the extended linear resource calculus λ^{fip} can be seen a pure *extension* of the FIP calculus; and the rules of the FIP calculus are a strict subset of λ^{fip}. In particular, FIP is exactly that subset of λ^{fip} which requires no dynamic reference counting or memory management at runtime. As a result, in the Perceus setting FIP functions can interact safely with any other function, executing in-place when possible and copying when necessary.
- In Section 6 we show a short performance evaluation of our particular implementation. It shows that `fip` algorithms are competitive to standard functional algorithms in Koka. This is somewhat expected since the standard algorithms can already avoid many allocations through the existing dynamic reuse as part of Perceus reference counting. If such dynamic reuse is disabled for the standard algorithms, `fip` functions tend to outperform with a larger margin.

Expressions:

$$
\begin{array}{lcll@{\qquad}lcll}
e & ::= & (v, \ldots, v) & \text{(unboxed tuple)} & v & ::= & x, y & \text{(variables)} \\
 & \mid & e\ e & \text{(application)} & & \mid & C^k\ v_1 \ldots v_k & \text{(constructor of arity } k\text{)} \\
 & \mid & f(e; e) & \text{(call)} & & & & \\
 & \mid & \mathsf{let}\ \overline{x} = e\ \mathsf{in}\ e & \text{(let binding)} & & & & \\
 & \mid & \mathsf{match}\ e\ \{\ \overline{p \mapsto e}\ \} & \text{(matching)} & p & ::= & C^k\ x_1 \ldots x_k & \text{(pattern)} \\
 & \mid & \mathsf{match!}\ e\ \{\ \overline{p \mapsto e}\ \} & \text{(destructive match)} & & & &
\end{array}
$$

$$
\Sigma \ ::= \ \varnothing \mid \Sigma, f(\overline{y}; \overline{x}) = e \quad \text{(recursive top-level functions with borrowed parameters } \overline{y}\text{)}
$$

Syntax:

$$
\begin{array}{lcll@{\qquad}lcl}
\overline{v} & \doteq & (v_1, \ldots, v_k) & (k \geqslant 1) & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \doteq & \mathsf{let}\ (x) = e_1\ \mathsf{in}\ e_2 \\
\overline{x} & \doteq & (x_1, \ldots, x_k) & (k \geqslant 1) & \lambda x_1, \ldots, x_k.\ e & \doteq & \lambda(x_1, \ldots, x_k).\ e \\
v & \doteq & (v) & \text{(unboxed singleton)} & & &
\end{array}
$$

Fig. 2. Syntax of the FIP calculus.

We have a full implementation in the Koka compiler [Leijen 2021,v2.4.2; Lorenzen et al. 2023b], and detailed proofs can be found in the technical report [Lorenzen et al. 2023a].

2 A LANGUAGE FOR FULLY IN-PLACE UPDATE

Figure 2 presents the syntax of the *fully in-place* FIP calculus. The syntax has been carefully chosen to be expressive enough to cover many interesting functions as shown in this paper, but at the same time restricted enough to be straightforward to analyze. Particular properties of our syntax are the inclusion of unboxed tuples, borrowed parameters, and the lack of general lambda expressions.

The syntax distinguishes between expressions e, and values v that cannot be evaluated further. Values are either variables or fully applied constructors C^k, taking k values as arguments. We often leave out the superscript k when not needed.

Unboxed tuples $(v_1, \ldots, v_k)$ are considered expressions, rather than values. In this way, we syntactically rule out that unboxed tuples may be passed as an argument to a constructor, causing them to become "boxed" (and allocated). Instead of enforcing this with a type system [Peyton Jones and Launchbury 1991], we use this syntactic restriction to enforce this property. By doing so, the check is simpler and allows us to specify the FIP calculus independent of its static semantics.

We often write just v or e for a singleton unboxed tuple (v), and write an overline to denote an unboxed tuple $(v_1, \ldots, v_k)$ as $\overline{v}$, or an unboxed tuple of variables $(x_1, \ldots, x_k)$ as $\overline{x}$. Since expressions always eventually evaluate to an unboxed tuple, the $\mathsf{let}\ \overline{x} = e_1\ \mathsf{in}\ e_2$ expression binds all components of the result unboxed tuple e_1 in $\overline{x}$.

There are no general lambda expressions. In general, closures need to be heap allocated if they contain free variables. To keep the FIP rules as simple as possible, we do not allow *arbitrary* lambda expressions. Instead, the global Σ environment holds all top-level functions f, which can be mutually recursive and passed as arguments. This makes our calculus essentially second-order. A top-level function is declared as $f(\overline{y}; \overline{x}) = e$, where $\overline{y}$ are the *borrowed* parameters, and $\overline{x}$ are the *owned* (unique) parameters. Just as in a let binding, the $\overline{y}$ and $\overline{x}$ bind the components of the unboxed tuples that are passed. A function is called by writing $f(e_1; e_2)$ with e_1 for the borrowed arguments, and e_2 for the owned arguments.

If there are no borrowed arguments, we sometimes write just $f(e_2)$. The syntax $e_1\ e_2$ is used for general application when the function to be called is not statically known. This happens when a function f is passed as an second-order argument itself, and in such case, e_2 is always passed as the owned parameter(s). We have two match expressions, the regular match, and the destructive

Evaluation order:

$$E ::= \Box \mid E\ e \mid \overline{v}\ E \mid f(E; e) \mid f(\overline{v}; E) \mid \mathsf{let}\ \overline{x}\ =\ E\ \mathsf{in}\ e$$
$$\mid \mathsf{match}\ E\ \{\ \overline{p\ \mapsto e}\ \} \mid \mathsf{match!}\ E\ \{\ \overline{p\ \mapsto e}\ \}$$

$$\frac{e_1 \longrightarrow e_2}{E[e_1] \longmapsto E[e_2]}\ \textsc{step}$$

Evaluation steps:

$$\begin{array}{llcll}
(\mathit{let}) & \mathsf{let}\ \overline{x}\ =\ \overline{v}\ \mathsf{in}\ e & \longrightarrow & e[\overline{x}{:=}\overline{v}] & \\
(\mathit{call}) & f(\overline{v_1}; \overline{v_2}) & \longrightarrow & e[\overline{y}{:=}\overline{v_1}, \overline{x}{:=}\overline{v_2}] & \text{with}\ f(\overline{y}; \overline{x})\ =\ e\ \in \Sigma \\
(\mathit{app}) & (f)\ \overline{v} & \longrightarrow & e[\overline{x}{:=}\overline{v}] & \text{with}\ f(; \overline{x})\ =\ e\ \in \Sigma \\
(\mathit{match}) & \mathsf{match}\ (C\ \overline{v})\ \{\ \overline{p\ \mapsto e}\ \} & \longrightarrow & e_i[\overline{y}{:=}\overline{v}] & \text{with}\ p_i\ =\ C\ \overline{y} \\
(\mathit{match!}) & \mathsf{match!}\ (C\ \overline{v})\ \{\ \overline{p\ \mapsto e}\ \} & \longrightarrow & e_i[\overline{x}{:=}\overline{v}] & \text{with}\ p_i\ =\ C\ \overline{x}
\end{array}$$

Fig. 3. Functional operational semantics.

match!. There is no difference in the functional semantics between the two, but in a heap semantics the destructive match can be used for reuse, and as we see in the FIP rules in Figure 4, it can only be used on owned parameters.

2.1 Functional Operational Semantics

Figure 3 gives the functional operational semantics for our calculus. This semantics does not yet use a heap. An evaluation context E is a term with a single occurrence of a hole $\Box$ in place of a subterm. For example, if $E\ =\ f(\overline{v}; \Box)$, then $E[(x, y)]$ is $f(\overline{v}; (x, y))$. Together with the STEP rule, E determines the evaluation order, where the hole denotes a unique subterm that can be reduced.

A small step reduction $e_1 \longrightarrow e_2$ evaluates e_1 to e_2. The reduction steps are standard except for always using unboxed tuples to substitute. We write $e[\overline{x}{:=}\overline{v}]$ for the capture-avoiding substitution of the distinct variables $\overline{x}\ =\ (x_1, \ldots, x_n)$ with the values $\overline{v}\ =\ (v_1, \ldots, v_m)$, where n must be equal to m and $\overline{x} \notin \mathsf{fv}(\overline{v})$. When we substitute in a function body, we write $e[\overline{x}{:=}\overline{v_1}, \overline{y}{:=}\overline{v_2}]$ to substitute all (distinct) variable $\overline{x}$ and $\overline{y}$ at once, where we again require a capture avoiding substitution with $\overline{x}, \overline{y} \notin (\mathsf{fv}(\overline{v_1})\ \cup \mathsf{fv}(\overline{v_2}))$.

When an expression e cannot be reduced further using STEP, then either e reduced to an unboxed tuple $\overline{v}$ and we are done, or we call the evaluation *stuck*. We have purposefully described the language and its dynamic semantics without a specific type system, but we can easily define standard Hindley-Milner typing rules [Hindley 1969; Milner 1978] to guarantee that well-typed programs never get stuck.

2.2 FIP: Fully In-Place

As defined, our functional semantics is very liberal and allows expressions that generally cause allocation, like $C\ x\ y$. Figure 4 specifies the FIP calculus rules that guarantee that the resulting programs can be evaluated without needing any (de)allocation. The statement $\Delta \mid \Gamma\ \vdash\ e$ means that under a borrowed environment Δ and owned environment Γ, the expression e is a well-formed FIP expression. The borrowed environment Δ is a *set* of borrowed variables which generally come from the borrowed parameters of a function f, or by borrowing in the LET rule. We write Δ, Δ' for the union of the sets Δ and Δ'. The owned environment Γ is a *multiset* of owned variables, and also *reuse credits*. Following Hofmann [2000a] we denote these as a diamond $\diamond_k$, signifying a credit of size k. We can append two owned environments Γ and Γ' as Γ, Γ'. Sometimes, we also write Δ, Γ to join a borrowed set with a multi-set Γ which does not contain reuse credits, where the result is the borrowed set Δ joined with the elements in Γ. Note that in the current rules, all variables in the Γ environment occur only once as we have no way to duplicate them. In Section 5 we generalize this to the full Perceus calculus.

$$
\begin{array}{lcll}
\Gamma & ::= & \varnothing \mid \Gamma, x \mid \Gamma, \diamond_k & \text{(owned environment)} \\
\Delta & ::= & \varnothing \mid \Delta, y & \text{(borrowed environment)}
\end{array}
$$

$$
\frac{}{\Delta \mid x \vdash x}\;\textsc{var}
\qquad\qquad
\frac{}{\Delta \mid \varnothing \vdash C}\;\textsc{atom}
$$

$$
\frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash (v_1, \ldots, v_n)}\;\textsc{tuple}
\qquad
\frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_k, \diamond_k \vdash C^k\, v_1 \ldots v_k}\;\textsc{reuse}
$$

$$
\frac{\overline{y} \in \Delta, \mathrm{dom}(\Sigma) \quad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash f(\overline{y}; e)}\;\textsc{call}
\qquad
\frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \quad \Delta \mid \Gamma_2, \Gamma_3, \overline{x} \vdash e_2 \quad \overline{x} \notin \Delta, \Gamma_2, \Gamma_3}{\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{let}\ \overline{x} = e_1\ \mathsf{in}\ e_2}\;\textsc{let}
$$

$$
\frac{y \in \Delta \quad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash y\, e}\;\textsc{bapp}
\qquad
\frac{y \in \Delta \quad \Delta, \overline{x}_i \mid \Gamma \vdash e_i \quad \overline{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma \vdash \mathsf{match}\ y\ \{\ C_i\, \overline{x}_i \mapsto e_i\ \}}\;\textsc{bmatch}
$$

$$
\frac{\Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma, \diamond_0 \vdash e}\;\textsc{empty}
\qquad
\frac{\Delta \mid \Gamma, \overline{x}_i, \diamond_k \vdash e_i \quad k = |\overline{x}_i| \quad \overline{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma, x \vdash \mathsf{match!}\ x\ \{\ C_i\, \overline{x}_i \mapsto e_i\ \}}\;\textsc{dmatch!}
$$

$$
\frac{}{\Vdash \varnothing}\;\textsc{defbase}
\qquad\qquad
\frac{\Vdash \Sigma' \quad \overline{y} \mid \overline{x} \vdash e}{\Vdash \Sigma', f(\overline{y}; \overline{x}) = e}\;\textsc{deffun}
$$

Fig. 4. Well-formed FIP expressions, where the multiplicity of each variable in Γ is 1.

The FIP rules ensure that variables in the owned environment Γ are used linearly (with some borrowing allowed in let). However, this is a syntactic property and we do not use a linear *type* system. This is much simpler to specify and implement, and also makes FIP independent of any particular type system used by a host language.

The linearity of the FIP calculus is apparent in the var rule, $\Delta \mid x \vdash x$ where we can only *consume* x if it is the only element of the owned environment Γ. Similarly, the tuple rule splits the owned enviroment in n distinct parts, Γ_i, and ensures well-formedness of each constituent value of the tuple, v_i, under the corresponding environment Γ_i. With the atom rule, $\Delta \mid \varnothing \vdash C$ we can return constructors without arguments which we consider allocation free. The owned environment must be empty here since our calculus is not affine: we cannot discard owned variables as that implies freeing a potentially heap allocated value (but in the next section we consider an extension of FIP that allows deallocation as well).

The only way to create a fresh constructor with $k \geqslant 1$ arguments, is through the reuse rule where we need to check well-formedness of each argument, but we also need a reuse credit $\diamond_k$ to guarantee that the needed space is available at evaluation time. The dmatch! rule creates such reuse credits: we can destructively match on an owned variable x to get a reuse credit $\diamond_k$ in each branch. In each branch the x is no longer owned though (but became a reuse credit instead). For simplicity we only allow matching on a variable in the rules, but we can always rewrite a user expression $\mathsf{match!}\ e\ \{ \ldots \}$ into $\mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{match!}\ x\ \{ .. \}$ where x is a fresh owned variable. Again, we do not allow freeing at this point, so reuse credits can only be consumed by the reuse rule or the empty rule for zero-sized reuse credits (when an atom is matched).

In contrast, the borrowed match bmatch can only match on borrowed variables and such match can only be used to inspect values without creating fresh reuse credits. Even though variables in the owned environment Γ cannot be discarded (i.e. freed) or duplicated (i.e. shared), we can temporarily *borrow* them. In the let rule the owned environment is split in three parts $\Gamma_1, \Gamma_2, \Gamma_3$.

$$
\begin{array}{rl}
\mathsf{S} ::= & \varnothing \mid \mathsf{S}, x \mapsto C^k\ x_1 \ldots x_k \ \mid \mathsf{S}, \diamond_k \\
E ::= & \Box \ \mid C^k\ x_1 \ldots E \ldots v_k \ \mid (x_1, \ldots, E, \ldots, v_n) \\
\mid & E\ e \mid x\ E \mid f(\overline{y}; E) \mid \mathsf{let}\ \overline{x}\ =\ E\ \mathsf{in}\ e \\
\mid & \mathsf{match}\ E\ \{\ \overline{p \mapsto e}\ \} \ \mid \mathsf{match!}\ E\ \{\ \overline{p \mapsto e}\ \}
\end{array}
\qquad
\dfrac{\mathsf{S} \mid e \ \longrightarrow_{\mathsf{s}} \mathsf{S}' \mid e'}{\mathsf{S} \mid E[e] \ \longmapsto_{\mathsf{s}} \mathsf{S}' \mid E[e']}\ \textsc{eval}
$$

$$
\begin{array}{llcll}
(\mathit{let}_s) & \mathsf{S} \mid \mathsf{let}\ \overline{x} = \overline{z}\ \mathsf{in}\ e & \longrightarrow_{\mathsf{s}} & \mathsf{S} \mid e[\overline{x}{:=}\overline{z}] & \\
(\mathit{call}_s) & \mathsf{S} \mid f(\overline{y}'; \overline{x}') & \longrightarrow_{\mathsf{s}} & \mathsf{S} \mid e[\overline{y}{:=}\overline{y}', \overline{x}{:=}\overline{x}'] & (f(\overline{y}; \overline{x}) = e\ \in \Sigma) \\
(\mathit{anon}_s) & \mathsf{S} \mid (f)\ \overline{x}' & \longrightarrow_{\mathsf{s}} & \mathsf{S} \mid e[\overline{x}{:=}\overline{x}'] & (f(\overline{y}; \overline{x}) = e\ \in \Sigma) \\
\\
(\mathit{reuse}_s) & \mathsf{S}, \diamond_k \mid C^k\ x_1 \ldots x_k & \longrightarrow_{\mathsf{s}} & \mathsf{S}, x \mapsto C^k\ x_1 \ldots x_k \mid x & (\mathsf{k} \geqslant 1,\ \mathsf{fresh}\ x) \\
(\mathit{atom}_s) & \mathsf{S} \mid C & \longrightarrow_{\mathsf{s}} & \mathsf{S}, x \mapsto C \mid x & (\mathsf{fresh}\ x) \\
(\mathit{bmatch}_s) & \mathsf{S},\ y \mapsto C^k\ \overline{z}\ \mid \mathsf{match}\ y\ \{\overline{p \to e}\} & \longrightarrow_{\mathsf{s}} & \mathsf{S},\ y \mapsto C^k\ \overline{z} \mid e_i[\overline{y}{:=}\overline{z}] & (p_i\ =\ C^k\ \overline{y}) \\
(\mathit{dmatch}_s) & \mathsf{S},\ x \mapsto C^k\ \overline{z} \mid \mathsf{match!}\ x\ \{\overline{p \to e}\} & \longrightarrow_{\mathsf{s}} & \mathsf{S}, \diamond_k \mid e_i[\overline{x}{:=}\overline{z}] & (p_i\ =\ C^k\ \overline{x})
\end{array}
$$

Fig. 5. Store semantics of FIP.

The Γ_1 and Γ_3 environments are passed to e_1 and e_2 respectively, but the Γ_2 environment is passed to e_2 as an owned enviroment, but also to e_1 as a borrowed environment! Since we *consume* Γ_2 in the derivation of e_2, we can consider them borrowed in the derivation of e_1. Note that we still need Γ_3 since Γ_2 is joined with the borrowed Δ environment and as such cannot contain any reuse credits (which can thus be included in Γ_3 instead).

The CALL rule is used for a function call $f(\overline{y}; e)$ where we can pass borrowed variables $\overline{y}$, and the owned argument e. To allow for passing functions, we can also pass a top-level function as part of $\overline{y}$. In the BAPP rule we can call such functions passed as a variable. Since we can only pass them as borrowed, we also only allow borrowed calls of the form $y\ e$. To prepare for an extension with full lambda expressions, we only allow owned arguments in a call, as already apparent in the operational semantics. Finally, we can check all top-level functions for well-formedness using the $\Vdash\ \Sigma$ rule. Any function $f \in \Sigma$ where $\Vdash\ \Sigma$ is considered *fully in-place*.

Implementing the check algorithmically is straightforward where the owned environment becomes synthesized. For let bindings we first check e_2 and use the synthesized Γ_2, Γ_3 to check e_1 (where Γ_3 only contains reuse credits). When merging synthesized environments we check if linearity is preserved. Since $\Delta \cap \Gamma$ is always empty, we can also infer whether to use a borrowed or destructive match and no such distinction is needed in the user facing syntax – we keep it in our calculus explicit though since we need the distinction in the store semantics.

2.3 Store Semantics

With the FIP calculus defined, we can now define another operational semantics. Figure 5 defines the *store semantics* where we evaluate using a fixed-size store S. The rules in the store semantics all adhere to an important invariant: each step of the evaluation should not allocate or deallocate memory. In this section, we establish a key result relating this store semantics with the functional operational semantics defined previously: under certain conditions, satisfied by all well-formed FIP programs, the store semantics and operational semantics coincide.

The store semantics uses an evaluation context E but this time a full evaluation goes to an unboxed tuple of variables $\overline{x}$ (instead of values $\overline{v}$). Any constructor is bound in the store S where every element is either a binding $x \mapsto^1 C^k\ x_1 \ldots x_k$ of size k, or a reuse credit $\diamond_k$ of size k.

Using the EVAL rule, we can reduce using small steps in the store semantics. The reduction rules have the form $\mathsf{S} \mid e \ \longrightarrow_{\mathsf{s}} \mathsf{S}' \mid e'$ where an e in a store S reduces to e' with a new store S'. The rules are similar to the earlier operational semantics but now we always substitute with variables instead of values. There are now two more rules for evaluating constructor values which are bound in the store. The ($reuse_s$) transition uses a reuse credit $\diamond_k$ in the store to apply the constructor, while ($atom_s$) allows atoms to be created freely. The ($bmatch_s$) and ($dmatch_s$) reductions differ, where the latter replaces the original constructor binding with a reuse credit of the same size.

Since our store semantics is destructive (in the reuse and dmatch rules), it can fail for expressions where the standard evaluation semantics would succeed. Even for expressions that are well-formed, the store semantics can fail if the initial store has internal sharing. If a shared variable is mutated in place, this would break referential transparency. Thus, we have to require that any variable is referred to just once in the store – we call this a *linear* store.

Definition 1. (*Store Soundness and Linearity*)
For a store S we write $\mathsf{dom}(\mathrm{S})$ to denote the set of variables x bound in S and write $\mathsf{rng}(\mathrm{S})$ to denote the set of values $C\ \overline{x}$ bound in S. A store is *sound* if all free variables in $\mathsf{rng}(\mathrm{S})$ are bound: $\mathsf{fv}(\mathsf{rng}(\mathrm{S})) \subseteq \mathsf{dom}(\mathrm{S})$. A store is *linear* if it is sound, and any variable x in $\mathsf{dom}(\mathrm{S})$ occurs at most once in the free variables of $\mathsf{rng}(\mathrm{S})$. By $\mathsf{roots}(\mathrm{S})$ we denote all reuse credits of S and the set of variables in $\mathsf{dom}(\mathrm{S})$ that do not occur in the free variables of $\mathsf{rng}(\mathrm{S})$.

On linear stores mutation is safe; in a reference counted setting such a store corresponds to a heap where all values have a reference count of one. We can now state the main soundness theorem. We write $[\mathrm{S}]\overline{x}$ to denote a substitution that recursively replaces variables by their bound value in S. We assume that we are given stores corresponding to the owned and borrowed values, but only require that the store of the owned values is linear. We can then show that the store evaluation leaves the borrowed values unchanged and only modifies the owned values:

Theorem 1. (*The store semantics is sound for well-formed FIP programs*)
If $\Delta \mid \Gamma \vdash e$ and given disjoint stores S_1, S_2 with $\Delta \subseteq \mathsf{dom}(S_1)$, S_1 sound, $\Gamma = \mathsf{roots}(S_2)$ and S_2 linear, then $[S_1, S_2]e \longmapsto^* \overline{v}$ implies $S_1, S_2 \mid e \longmapsto^*_{\mathrm{s}} S_1, S_3 \mid \overline{x}$ where $[S_3]\overline{x} = \overline{v}, \overline{x} = \mathsf{roots}(S_3)$ and S_3 is linear.

This is a strong theorem and the proof is quite involved (see App. B of the tech. report), both due to destructive update and the ability to match on variables temporarily borrowed in the LET rule. As a corollary, we can now see that any FIP expression can run on the store semantics if we use a store containing the necessary reuse credits, i.e. we give it enough space to allocate upfront:

Corollary 1.
If $e \longmapsto^* \overline{v}$ and $\varnothing \mid \overline{\diamond_k} \vdash e$, then $\overline{\diamond_k} \mid e \longmapsto^*_{\mathrm{s}} \mathrm{S} \mid \overline{x}$ and $[\mathrm{S}]\overline{x} = \overline{v}$.

We can define the *size* of a store by adding the sizes of all bindings within it. Since atoms and empty reuse credits have size zero, they do not contribute to the size of the store.

Definition 2.
The size $|\mathrm{S}|$ of a store S is: $\quad |\varnothing| = 0 \quad |\mathrm{S}, \diamond_k| = |\mathrm{S}| + k \quad |\mathrm{S}, x \mapsto C^k\ x_1 \ldots x_k| = |\mathrm{S}| + k.$

With this definition, we can immediately see that the size of the store does not change in any reduction of the store semantics. As such, FIP programs can reduce *in-place* without any (de)allocation:

Theorem 2. (*A FIP program reduces in-place.*)
For any $\mathrm{S} \mid e \longmapsto^*_{\mathrm{s}} \mathrm{S}' \mid e'$, we have $|\mathrm{S}| = |\mathrm{S}'|$.

2.4 FBIP: Allowing Deallocation

Our basic FIP calculus is quite strict and allows neither allocation nor deallocation. We can easily extend it though to allow deallocation. Figure 6 describes the FBIP calculus as an extension of

Extended Syntax:

$$e \;::=\; \ldots \mid \mathsf{drop}\ x;\ e \mid \mathsf{free}\ k;\ e$$

Extended evaluation rules:

$$\begin{array}{llll} (\mathit{dcon}_s) & \mathrm{S}, x \mapsto C^k\ \overline{x} & \mid \mathsf{drop}\ x;\ e & \longrightarrow_{\mathrm{s}} \quad \mathrm{S} \mid \mathsf{drop}\ \overline{x};\ e \\ (\mathit{free}_s) & \mathrm{S}, \diamond_k & \mid \mathsf{free}\ k;\ e & \longrightarrow_{\mathrm{s}} \quad \mathrm{S} \mid e \end{array}$$

$$\frac{\Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma, x \vdash \mathsf{drop}\ x;\ e}\ \textsc{drop} \qquad\qquad \frac{\Delta \mid \Gamma \vdash e \quad k \geqslant 1}{\Delta \mid \Gamma, \diamond_k \vdash \mathsf{free}\ k;\ e}\ \textsc{free}$$

Fig. 6. The FBIP calculus extends FIP with deallocation.

the FIP calculus with deallocation, where the syntax is extended with drop x; e to drop an owned variable x, and free k; e to free a reuse credit of size k.

The DROP rule consumes a variable x from the owned environment. Since the multiplicity of all elements in Γ is still one, this asserts that x is no longer an element of Γ. Similarly, the FREE rule allows discarding a reuse credit.

The operational semantics is also extended with two new reductions for dropping a bound contructor and freeing a reuse credit. With these new rules and reductions for deallocation, the soundness theorem 1 continues to hold (see App. B of the tech. report). Again, we can immediately see that the store semantics now only allows deallocation:

Theorem 3. (*A FBIP program can only deallocate.*)
For any $\mathrm{S} \mid e \longmapsto_{\mathrm{s}}^{*} \mathrm{S}' \mid e'$ with the deallocation rules, we have $|\mathrm{S}| \geqslant |\mathrm{S}'|$.

In our implementation the `fbip` keyword checks if the function is well-formed in the FBIP calculus.

2.5 Stack Safe FIP

So far, our FIP calculus has allowed us to bound the heap space of the program—but what about the stack space? If we only seek to bound allocations, we could choose to leave it unbounded. In practice, however, the stack space matters: when compiling FIP programs to C we have to assume a relatively small stack and even in a garbage-collected setting growing the stack is not free. To ensure that the stack is bounded, we require two modifications to the calculus. We assume that any function f is defined as part a of mutually recursive group $\overline{f}$ (which might consist of just f or more functions). In the CALL rule we then require two additional conditions. Firstly, Σ contains only functions defined before or as part of the current mutually recursive group. Secondly, we constrain the mutually recursive groups $\overline{f}$ by requiring that any recursive calls within this group are tail-recursive. Formally, all function definitions $f \in \overline{f}$ need to be of the form $f(\overline{y}; \overline{x}) \;=\; \mathcal{T}[\overline{f}]$:

$$\begin{array}{rl} \mathcal{T}[\overline{f}] \;::=\; e_0 & \mid f_i(e_0; e_0) \;\mid \mathsf{let}\ \overline{x} \;=\; e_0\ \mathsf{in}\ \mathcal{T}[\overline{f}] \\ \mid & \mathsf{match}\ e_0\ \{\ \overline{p_i \mapsto \mathcal{T}_i[\overline{f}]}\ \}\ \mid \mathsf{match!}\ e_0\ \{\ \overline{p_i \mapsto \mathcal{T}_i[\overline{f}]}\ \}\ \mid \mathsf{drop}\ \overline{x};\ \mathcal{T}[\overline{f}]\ \mid \mathsf{free}\ k;\ \mathcal{T}[\overline{f}] \end{array}$$

wher $\overline{f} \not\pitchfork \mathsf{fv}(e_0)$. In the CALL rule, one can pass functions from Σ to the called function. By the first constraint, these functions can only be defined before or mutually recursive with the current definition. In the tail-context we further require that any functions passed as arguments are also not in the mutually recursive group. Thus, we can only pass functions that were defined strictly before the current definition. We call the FIP calculus extended with these requirements FIP$^{\mathrm{S}}$, in which the stack usage is always bounded. The `fip` keyword in our implementation checks if a function is a well-formed FIP$^{\mathrm{S}}$ function.

To show this formally, we use the size of the evaluation context as a proxy for the stack size. We write $|e|$ for the depth of an expression e and $|E|$ for the depth of an evaluation context. We fix a signature Σ and denote by $|e_{\max}|$ the maximum depth of an expression bound in Σ. Then we have:

Theorem 4. (*A FIP program uses constant stack space*)
Let Σ be fully-in-place such that for all functions f in Σ that are mutually recursive with $\overline{f}$, we have $f(\overline{y};\overline{x}) = \mathcal{T}[\overline{f}]$. At any intermediate evaluation step $S \mid f(\overline{y};\overline{x}) \longmapsto_s^* S' \mid E[e']$, we have $|E| \leqslant |e_{\max}| \cdot |\Sigma|^2$.

This then yields our stack size bound of $|\Sigma|^2$. The additional factor of $|e_{\max}|$ describes the maximum size of the evaluation context within each function. In practice, we would not allocate a stack frame for these parts of the evaluation context. In a first-order context we would expect a stack bound of $|\Sigma|$ (where any function can call functions defined before it). However, in a second-order calculus, any function can call anonymous functions defined *after* it which adds another factor of $|\Sigma|$ (and see App. C of the tech. report for a detailed proof).

3 FULLY IN-PLACE TRAVERSALS OVER POLYNOMIAL DATATYPES

A classic example of a fully in-place algorithm is the in-place traversal of a binary tree [Reinking, Xie et al. 2021]. Consider a binary tree with all the values at the tips:

```
type tree<a>
  Bin( left: tree<a>, right: tree<a> )
  Tip( value: a )
```

Similar to our earlier splay tree in Section 1, we can again define a zipper to help traverse the tree in-order:

```
type tzipper<a,b>
  Top
  BinL (up : tzipper<a,b>, right: tree<a>)
  BinR (left : tree<b>, up : tzipper<a,b>)
```

A `tzipper<a,b>` stores fragments of the input tree in-order: those subtrees we have not yet visited are stored using the `BinL` constructor; the subtrees we have already visited are stored in the `BinR` constructor. We can now map over the tree in-order without using heap- or stack space by reusing the `tzipper` nodes. To define the tree map function, we begin by repeatedly stepping `down` through the input tree to the leftmost tip. Each subtree we have not yet visited, is accumulated in a `BinL` constructor. Once we hit the leftmost leaf, we apply the argument function `f`, and work our way back up, recursively processing any unvisited subtrees:

```
fip fun down( t : tree<a>, ^f : a -> b, ctx : tzipper<a,b> ) : tree<b>
  match! t
    Bin(l,r)  -> down( l, f, BinL(ctx,r) )   // go down the left spine, remember to visit r later
    Tip(x)    -> app( Tip(f(x)), f, ctx)     // start upwards along the zipper

fip fun app( t : tree<b>, ^f : a -> b, ctx : tzipper<a,b> )  : tree<b>
  match! ctx
    Top         -> t
    BinR(l,up)  -> app( Bin(l,t), f, up)     // keep going up rebuilding the tree
    BinL(up,r)  -> down( r, f, BinR(t,up) )  // go down a right side

fip fun tmap( t : tree<a>, ^f : a -> b ) : tree<b>
  down(t,f,Top)
```

The mutually tail-recursive `app` and `down` functions are fully in-place since each matched `Bin` can be paired with a `BinL`, each `BinL` with a `BinR`, and finally each `BinR` with a `Bin` again. The definition of `tmap` may seem somewhat involved, yet consider writing this function in an imperative language, without using extra stack- or heap space, mutating pointers throughout the tree.

Seeing how we can write a map over a binary tree as a FIP function, we may ask if this is possible perhaps for any simple algebraic datatype that can be expressed as a sum of products. It turns out this is indeed the case, and we show this in two steps: first we show in the next subsection a general method for rewriting programs that are tail-recursive *modulo reusable contexts* (TRMReC) such that they are fully in-place. Then, we show how we can generically derive a `map` function for any polynomial inductive datatype to which our TRMReC translation can be applied.

3.1 Tail Recursion Modulo Reusable Defunctionalized CPS Contexts

While our FIP `tmap` function may seem very different from a standard map over trees, it turns out that it actually corresponds to the defunctionalized CPS [Danvy 2008; Reynolds 1972] version of the standard `tmap` function:

```
fun tmap(t : tree<a>, ^f : a -> b) : tree<a>
  match! t
    Bin(l,r)  -> val l’ = tmap(l,f) in val r’ = tmap(r,f) in Bin(l’, r’)
    Tip(a)    -> Tip(f(a))
```

Let us focus on the first branch, where a CPS-translation yields the following closures:

```
Bin(l,r) -> tmap(l,f, fn(l’){ tmap(r,f, fn(r’){ k(Bin(l’, r’)) }) })
```

Comparing with our `tzipper` type, we can identify `Top` with the identity function, `BinR` with the inner closure (`fn(r’) k(Bin(l’, r’))`), and `BinL` with the outer closure – the zipper is just the defunctionalization of the closures:

```
fn(x) x                                       ===   Top        -> t
fn(r’) k(Bin(l’,r’))                          ===   BinR(l’,k) -> app( Bin(l’,t), f, k )
fn(l’) tmap(r,f,fn(r’){ k(Bin(l’,r’)) }) })   ===   BinL(k,r)  -> down(r,f,BinR(t,k))
```

The arguments `r’` and `l’` to the closures correspond to the tree `t`, the `down` function to the transformed `tmap` function, and `app` applies the defunctionalized continuation `k` to the new tree. As shown by Danvy [2022], this defunctionalized CPS-transformation applies widely and can transform many programs from direct-style to tail-recursive form. But are these techniques also applicable when writing FIP programs? Sobel and Friedman [1998] show that it is always possible to reuse the zipper for the result in all anamorphisms. In fact, in the above translation, it is even possible to reuse the initial tree to construct the zipper.

Using this insight, we can give a general translation to tail-recursive programs that is guaranteed fully in-place. It is inspired by the defunctionalized CPS-translation, but we have to make several small adjustments to make it work. For example, notice how the borrowed function `f` is not included in the zipper, but instead passed directly to `app`. This is crucial, since we can not store a borrowed value inside a data structure. Figure 7 shows the formal transformation, based on the general framework of tail recursion *modulo context* as shown recently by Leijen and Lorenzen [2023].

Starting with a function $f(\overline{y};\overline{x}) \;=\; e$, we first define the zipper by creating a constructor H for the identity and one constructor Z_i each for each evaluation context E_i in e that contains a recursive call to f. Each constructor carries the free variables $\overline{z}_i$ of its evaluation context and the link to the parent zipper z'. We transform f into $f'(\overline{y};\overline{x},z)$ and provide an $app(\overline{y};z,\overline{x}')$ function, where we ensure that both receive the same borrowed variables $\overline{y}$. The calls to f' receive the current zipper as an extra argument z and is defined by the translation $[\![\, e \,]\!]_z$ below. The *app* function matches on the current zipper (as before) and resumes execution in the relevant (transformed) evaluation context.

The transformation follows defunctionalized CPS contexts of the TRMC framework [Leijen and Lorenzen 2023,Sec. 4.3] with the (*tctx*), (*base*), (*tail*), and (*ectx*) rules. If we encounter a tail context $\mathbb{T}$, we continue the transformation in every hole using the (*tctx*) rule. When we encounter a term e_0 which has no recursive calls, the (*base*) rule inserts a call to *app* to apply the result of e_0 to the continuation stored in z. If we encounter a tail-call we simply leave it as is. Finally, if we find a call in an evaluation context, we turn it into a tail-call by storing the free variables of E_i and the current zipper. Notice that we cannot have a tail-context nested in an evaluation context as the translation assumes that programs are in A-normal form [Flanagan et al. 1993].

Translating f to a tail-recursive f' function:

$$
\begin{array}{lcl}
f(\overline{y};\overline{x}) & = & e \\
f'(\overline{y};\overline{x},z) & = & \llbracket\, e \,\rrbracket_z \\
\mathit{zipper} & = & \mathsf{H} \;\mid Z_1\;\overline{z}_1\;z' \;\mid \ldots \;\mid Z_n\;\overline{z}_n\;z'
\end{array}
$$

Applying the zipper:

$$
\begin{array}{ll}
\mathit{app}(\overline{y};z,\overline{x}') = \text{match! } z & \\
\qquad \mathsf{H} \rightarrow \overline{x}' & \\
\qquad Z_1\;\overline{z}_1\;z' \rightarrow \llbracket\, E_1[\overline{x}'] \,\rrbracket_{z'} & \\
\qquad \ldots & \\
\qquad Z_n\;\overline{z}_n\;z' \rightarrow \llbracket\, E_n[\overline{x}'] \,\rrbracket_{z'} &
\end{array}
$$

Tail recursive contexts with $f \notin \mathsf{fv}(e_0)$:

$$\mathbb{T} ::= \Box \;\mid \text{let } \overline{x} \;=\; e_0 \text{ in } \mathbb{T} \;\mid \text{match } e_0 \;\{\, \overline{p_i \;\mapsto \mathbb{T}_i} \,\} \mid \text{match! } e_0 \;\{\, \overline{p_i \;\mapsto \mathbb{T}_i} \,\} \;\mid \text{drop } \overline{x};\; \mathbb{T} \;\mid \text{free } k;\; \mathbb{T}$$

Tail recursion translation for a function f with zipper z:

$$
\begin{array}{llcll}
(\mathit{tctx}) & \llbracket\, \mathbb{T}[e] \,\rrbracket_z & = & \mathbb{T}[\llbracket\, e \,\rrbracket_z] & \\
(\mathit{base}) & \llbracket\, e_0 \,\rrbracket_z & = & \mathsf{app}(\overline{y};z,e_0) & \text{where } f \;\notin \mathsf{fv}(e_0) \\
(\mathit{tail}) & \llbracket\, f(\overline{y};\; \overline{x}') \,\rrbracket_z & = & f'(\overline{y};\; \overline{x}',\; z) & \\
(\mathit{ectx}) & \llbracket\, E_i[f(\overline{y};\; \overline{x}')] \rrbracket_z & = & f'(\overline{y};\; \overline{x}',\; Z_i^k\;\overline{z}_i\;z) & \text{where } \overline{y} \mid \diamond_k, \overline{z}_i \vdash E_i[\Box] \text{ and } k = |\overline{z}_i| + 1
\end{array}
$$

Fig. 7. TRMReC: tail-recursive modulo reusable contexts.

So far this transformation describes just how to enable tail recursion on general defunctionalized CPS contexts but the result is not yet guaranteed to be FIP. To guarantee that reuse applies we need to preserve the side-condition on (ectx), which ensures that E_i does not depend on borrowed variables other than $\overline{y}$ (which could not be stored in an accumulator) and that there is a space credit of the appropriate size for z and the free variables $\overline{z}_i$ of E_i.

If this condition is met, this transformation yields a tail-recursive, fully in-place program:

Theorem 5. (*The TRMReC transformation is sound.*)
Let f be a function with $\overline{y} \mid \overline{x} \vdash f(\overline{y};\overline{x})$ and $f(\overline{v}_1;\overline{v}_2) \longrightarrow^* \overline{w}$. If it can be transformed into f', then $\overline{y} \mid \overline{x}, z \vdash f'(\overline{y};\overline{x},z)$ and $\overline{y} \mid z, \overline{x} \vdash \mathsf{app}(\overline{y};z,\overline{x})$ and $f'(\overline{v}_1;\overline{v}_2,\mathsf{H}) \longrightarrow^* \overline{w}$.

See App. D of the tech. report for the proof. We also generalize this theorem to handle recursive calls with varying borrowed arguments and the case where $k \geq |\overline{z}_i| + 1$ (as common in folds) or more than one reuse credit is available. Clearly, this translation can apply to the `tmap` introduced at the start of this section. We just have to check the side condition:

- For $E_1 :=$ let $l' \;=\; \Box$ in let $r' \;=\;$ tmap$(f;\; r)$ in Bin $l'\; r'$ we have $f \mid \diamond_2, r \vdash E_1[\Box]$.
- For $E_2 :=$ let $r' \;=\; \Box$ in Bin $l'\; r'$ we have $f \mid \diamond_2, l' \vdash E_2[\Box]$.

Thus the condition is fulfilled and the translation succeeds.

3.2 Schorr-Waite Tree Traversals

Using the translation in the previous section we can now generalize the `tmap` function to any polynomial inductive datatype. Following the approach by van Laarhoven [2007] to generically derive functors in Haskell, we use a generic macro $\$\mathsf{map}_\tau$ to define a (non tail-recursive) *map* function for any type $T\;\alpha$ in a straightforward way, where we match on each constructor in T and call the \$map macro on each of the fields:

$$
\begin{array}{l}
\mathit{map}(f;x) \;=\; \text{match! } x \\
\quad C^k\;(x_1 : \tau_1) \ldots (x_k : \tau_k) \rightarrow \\
\qquad \text{let } y_1 \;=\; \$\mathsf{map}_{\tau_1}(f;x_1) \text{ in} \\
\qquad \ldots
\end{array}
$$

$$
\begin{array}{lcll}
\$\mathsf{map}_{T\;\alpha}(f;x_i) & = & \mathit{map}(f;x_i) & \\
\$\mathsf{map}_{\alpha}(f;x_i) & = & f(x_i) & \\
\$\mathsf{map}_{\tau}(f;x_i) & = & x_i & \text{otherwise}
\end{array}
$$

$$\begin{array}{ll} \text{let } y_k \;=\; \$\mathsf{map}_{\tau_k}(f; x_k) \text{ in} & \\ C^k\ y_1 \ldots y_k & E_i \;=\; \text{let } y_i \;=\; \square \text{ in} \ldots \text{in } C^k\ y_1 \ldots y_k \\ \ldots & f \mid \diamond_k, y_1, \ldots, y_{i-1}, x_{i+1}, \ldots, x_k \vdash E_i[\square] \end{array}$$

The \$map$_\tau$ macro dispatches on the *type* of its argument. if x has type $T\ \alpha$ it generates a recursive call $map(f; x)$ if x has type α, it generates a call $f(x)$; otherwise it leaves the argument unchanged. In this definition, all recursive calls to *map* happen in \$*map*, each application of which is in an evaluation context E_i where the side-condition holds. Thus, we can apply the TRMReC transformation of the previous section and automatically obtain a tail-recursive fully in-place version of *map*.

Why does reuse work so naturally here? Part of the solution seems to be that the link to the parent is stored together with the other free variables. In contrast, McBride [2008] defines a generic *fold* function which is not fully in-place since it stores the defunctionalized continuations on a stack. Nevertheless, as McBride [2001] shows, the defunctionalized continuations correspond to the (generalised) *derivative* of a regular type $T\ \alpha$. For every constructor C with k recursive subtrees, the derivative datatype has k constructors, one for each possible continuation. Reuse then arises naturally, as the constructors of the derivative and original datatype can line up perfectly.

In the literature on imperative algorithms, these traversals that use no extra stack space except for direction hints (as encoded in the constructors of the zipper datatype), are known as Schorr-Waite traversals [Schorr and Waite 1967]. Effectively, we can thus derive a Schorr-Waite traversal for any polynomial algebraic datatype and use the reuse analysis of the FIP calculus to compile it to the corresponding imperative code. This is remarkable as imperative Schorr-Waite traversals are notoriously difficult to get right or prove correct. In his famous work on separation logic, Reynolds [2002] writes:

> The most ambitious application of separation logic has been Yang's proof of the Schorr-Waite algorithm for marking structures that contain sharing and cycles.

Of course, our construction cannot handle cycles, but rather shows the traversal of trees (or any polynomial inductive datatype in general). The tree traversal by itself, however, is already quite complicated and has become a benchmark for verification frameworks [Loginov et al. 2006; Walker and Morrisett 2000]. Furthermore, our translation also shows that the Schorr-Waite tree traversal is equivalent to a stack-based depth-first traversal (like the standard `tmap`). This was already shown by Yang [2007] in the context of separation logic, but that required more advanced methods than straightforward induction.

4 FURTHER EXAMPLES OF FULLY IN-PLACE ALGORITHMS

Many common functions used in functional programming are already FIP in their standard definition, like `map` or `reverse`. In this section we want to present some advanced examples that test the limits of what is possible. Along the way, we see several techniques that may be of general use for designing algorithms in a language with in-place reuse. These include passing reuse credits to functions, padding constructors and the `partition` datastructure. Full listings of the examples in this section can be found in App. A of the tech. report.

4.1 Imperative Red-Black Tree Insertion, Functionally

Can insertion into a red-black tree be FIP? The traditional implementation of red-black trees, due to Okasaki [1999], can indeed be written to use all its arguments in-place. However, it occasionally has to rebalance the result of the recursive call and thus uses stack space linear in the depth of the tree. However, we can avoid this by using a zipper (as in Section 3) and balancing while reconstructing the tree from the zipper. Surprisingly, this yields a functional implementation which is almost identical to the imperative red-black tree insertion algorithm described in the popular

“Introduction to Algorithms” textbook [Cormen et al. 2022]. We first define the tree and its zipper:

```
type color { Red; Black }
type tree<k,v>
  Node(c : color, l : tree<k,v>, k : k, v : v, r : tree<k,v>)
  Leaf
type accum<k,v>
  Done
  NodeL(c : color, l : accum<k,v>, k : k, v : v, r : tree<k,v>)
  NodeR(c : color, l : tree<k,v>, k : k, v : v, r : accum<k,v>)
```

We can insert a value into the tree by recursing into the left- or right-subtree until we either find the key with an existing value or a leaf. In the latter case, we will have to allocate a new node for key and value. During the recursion, we turn the nodes of the tree into the accum zipper. At the end, we thus have a subtree with our new value and a zipper. We create a rebalanced red-black tree by calling the fixup function below on them.

But how do we write fixup? Thankfully, we can translate it almost verbatim from the rb-insert-fixup procedure in section 13.3 of Cormen et al. [2022]! We present the code here to illustrate that this function is FIP and closely follows its imperative counterpart. A detailed explanation can be found in [Cormen et al. 2022]. The function distinguishes three cases (marked on the left of the function definition) that correspond to the three cases in the textbook implementation. Case one translates directly (even if we have it twice). The second case is the most complicated, rotating an inner part left, before rotating the outer part right. In case three, it is possible to stop fixup (by calling rebuild defined below) as the parent node was colored black.

```
fip fun fixup( zp : accum<k,v>, z : tree<k,v> ) : tree<k,v>
  match! zp
    NodeL(Red,zpp,zpk,zpv,zpr) -> match zpp
      NodeL(Black,zppp,zppk,zppv,y) ->
        if is-red(y)
        then fixup(zppp,Node(Red,Node(Black,z,zpk,zpv,zpr),zppk,zppv,y.set-black))       // (1)
        else rebuild(zppp,right-rotate(Node(Red,Node(Black,z,zpk,zpv,zpr),zppk,zppv,y))) // (3)
    NodeR(Red,zpl,zpk,zpv,zpp) -> match zpp
      NodeL(Black,zppp,zppk,zppv,y) ->
        if is-red(y)
        then fixup(zppp,Node(Red,Node(Black,zpl,zpk,zpv,z),zppk,zppv,y.set-black))       // (1)
        else rebuild(zppp,right-rotate(                                                  // (2)
          Node(Red,left-rotate(Node(Red,zpl,zpk,zpv,z.set-black)),zppk,zppv,y)))
    // and cases as above with "left" and "right" interchanged
    _ -> rebuild(zp, z)
```

We have left some functions unspecified: set-black sets the color of a node to black. is-red returns a boolean indicating whether the given node is red. We have to make its argument borrowed so that we apply r2 to it. The left rotation is as usual (and the right rotation its mirror image).

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

fip fun left-rotate(t : tree<k,v>) : tree<k,v>
  match! t { Node(c,l,k,v,Node(c1,l1,k1,v1,r1)) -> Node(c1,Node(c,l,k,v,l1),k1,v1,r1); t’ -> t’ }
```

The last remaining function is rebuild, which is called when balancing is finished. In the imperative implementation, this simply marks the root black and returns the root. But in our version, the root is now hidden in the zipper and we have to rebuild the tree from the zipper (without balancing) to access the root:

```
fip fun rebuild(z : accum<k,v>, t : tree<k,v>)
  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.set-black
```

In practice, the imperative version benefits from not having to rebuild the tree (and fixup considers some cases specifically to be able to enter `rebuild` earlier). Thus we can not quite achieve the same efficiency in a functional version. However, our version can also be used persistently (see Section 5) and might be easier to understand.

4.2 Sorting Lists In-Place

Is it possible to run merge sort in-place? The traditional functional implementation first turns each element into a singleton list. A singleton list is obviously sorted, so we can now pairwise merge such sorted lists. Finally, we end up with just one sorted list, which we extract:

```
[4,3,2,1] -> [[4],[3],[2],[1]]    // create sorted singleton lists
  -> [[3,4],[1,2]] -> [[1,2,3,4]] // merge pairs of sorted lists
  -> [1,2,3,4]                    // extract sorted list
```

Here, the output takes up just as much space as the input - so a fully in-place implementation might be possible. But in the first step, many singleton lists are created for which no reuse tokens are available, so the traditional implementation is *not* fully in-place.

However, we can make it FIP by using a tailor made datastructure, that can store the sorted sublists while taking up exactly the same amount of space as the original list. Our `partition` is a list partitioned into sublists, which are either `One`s or `Sub`s of at least two elements. We exploit that we have at least two elements by storing two elements in the last cell of a `list2`.

```
type partition<a>
  Sub(list : list2<a>, tail : partition<a>)
  One(elem : a, tail : partition<a>)
  End
type list2<a>
  Cons2(x : a, tail : list2<a>)
  Nil2(x : a, y : a)
```

Considering the memory usage, we see that a `list2` can store n elements in $n-1$ cells. As a result, a `partition` of n elements uses just as much space as a list of n elements, while also keeping information about the partitioning. With that in hand, we can implement an in-place list mergesort:

```
Cons(4,Cons(3,Cons(2,Cons(1,Nil))))        // start with unsorted list
  -> One(4,One(3,One(2,One(1,End))))       // created sorted singleton lists
  -> Sub(Nil2(3,4),Sub(Nil2(1,2),End))     // merge pairs of sorted lists
  -> Sub(Cons2(1,Cons2(2,Nil2(3,4))),End)  // ... until only one list2 is left
  -> Cons(1,Cons(2,Cons(3,Cons(4,Nil))))   // convert back to list
```

This datastructure is also helpful to implement a quicksort that can not run out of stack. The typical functional but in-place implementation is [Baker 1994a; Hofmann 2000b; Hudak 1986]:

```
fbip fun quicksort(xs : list<a>)
  match! xs
    Nil -> Nil
    Cons(pivot, xx) ->
      val (lo, hi)  = split(pivot, xx)                  // split xx into (lo,hi) in-place
      val (lo',hi') = (quicksort(lo),quicksort(hi))      // sort the sublists
      append(lo', Cons(pivot, hi'))
```

This code is not FIP because the stack usage is not bounded and might grow linear with the length of the input list. Of course, we could apply the defunctionalized CPS-transformation (see Section 3), but our side-condition fails:

```
    Cons(pivot, xx) ->                       // reuse credit of size 2 available
      val (lo, hi) = split(pivot, xx)
      quicksort'(lo, Z1(pivot, hi, zipper))  // reuse credit of size 3 needed
```

The problem here is that the zipper needs to store `pivot,hi` and the parent `zipper`, which requires more space than we have available. This is because, in a stack-safe quicksort, the zipper needs to keep track of all the pivots and `hi` lists that still need to be sorted. However, we can use a `partition` structure as the zipper where we store the pivots as singletons and the `hi` either not at all (if `hi` is empty), as a singleton (if `hi` is a one-element list) or else as a `list2`. We can pass the parent zipper into the `split` function, which now returns a list `lo` and a `partition hi` which includes the zipper. Then we obtain a fully in-place solution:

```
Cons(pivot, xx) ->                   // reuse credit of size 2 available
  val (lo, hi) = split(pivot, xx, zipper)
  quicksort'(lo, One(pivot, hi))       // reuse credit of size 2 needed
```

4.3 Finger Trees

Finally, as an advanced example, we want to consider finger trees [Claessen 2020; Hinze and Paterson 2006], an efficient functional implementation of sequences. Yet, at first glance the `cons` function on finger trees does not appear to be FIP: only the `More` constructors can be reused as the other datatypes do not match up for reuse. We can fix this, however, by *padding* all constructors with a dummy atom `Pad` so that they all have three slots.

```
fun cons(x : a, s : seq<a>) : seq<a>
  match! s
    Empty -> Unit(x, Pad, Pad)
    Unit(y, _, _) -> More(One(x, Pad, Pad), Empty, One(y, Pad, Pad))
    More(One(y, _, _), q, u) -> More(Two(x, y, Pad), q, u)
    More(Two(y, z, _), q, u) -> More(Three(x, y, z), q, u)
    More(Three(y, z, w), q, u) -> More(Two(x, y, Pad), cons(Pair(z, w, Pad), q), u)
```

We have gotten rid of all deallocations since all constructors on the left of `->` can be paired with one to the right. But we still have allocations in the `Empty`, `Unit` and `More(Three)` cases. Even worse, the `cons` can recurse up to $O(\log n)$-times in the `More(Three)` case and require a new memory cell each time, so this function is not `fip(n)` or `fbip(n)`. However, this case is very unlikely as the amortized complexity analysis of finger trees shows that `cons` only recurses $O(1)$-times on average and thus only uses a constant amount of memory.

Therefore, we pair a finger tree `seq<a>` with a buffer which contains exactly the memory needed. Our buffer is just a padded list of size 3, which makes it available for reuse with the rest of the finger tree.

```
type buffer { BEmpty; BCons(next : buffer, b : pad, c : pad) }
```

We then pass the necessary reuse credits of size 3 to `cons`, which we either use to create a new cell in the finger tree or fill up the buffer. If we recurse into `cons`, we draw the necessary memory back from the buffer. Then we only need to ensure that we pass enough credits so that the buffer is never empty. Inserting two elements `x,y` into an empty finger tree yields `More(One(x, Pad, Pad), Empty, One(y, Pad, Pad))`, so it would seem that we need to pass at least two credits each. But that would mean that we need $6n$ space to represent n elements in a finger tree! We can do better by specializing `More(One)` as `More0` to represent the two-element list as `More0(x, Empty, One(y, Pad, Pad))`. With this modification, it suffices to pass in a single reuse credit per element for a space overhead of $3n$ space, which is close to the $2n$ factor of singly-linked lists. Our `cons` function than takes a reuse credit `unit3` and becomes:

```
fip fun cons(x : a, u3 : unit3, s : seq<a>, b : buffer) : (seq<a>, buffer)
  match! s
    Empty -> (Unit(x, Pad, Pad), b)
    Unit(y, _, _) -> (More0(x, Empty, One(y, Pad, Pad)), b)
    More0(y, q, u) -> (More(Pair(x, y, Pad), q, u), b)
```

Extended syntax :

$$\begin{array}{lll} v & ::= & \ldots \mid \lambda^{\overline{z}}\,\overline{x}.\ e \quad (\text{lambda with } \overline{z} \ = \ \mathsf{fv}(\lambda\overline{x}.\ e)) \\ e & ::= & \ldots \mid \mathsf{dup}\ x;\ e \mid \mathsf{dropru}\ x;\ e \mid \mathsf{alloc}\ k;\ e \end{array}$$

Extended evaluation steps :

$$\begin{array}{llll} (\mathit{beta}) & (\lambda^{\overline{z}}\,\overline{x}.\ e)\ \overline{v} & \longrightarrow & e[\overline{x}{:=}\overline{v}] \\ (\mathit{dup}) & \mathsf{dup}\ x;\ e & \longrightarrow & e \\ (\mathit{dropru}) & \mathsf{dropru}\ x;\ e & \longrightarrow & e \\ (\mathit{alloc}) & \mathsf{alloc}\ k;\ e & \longrightarrow & e \end{array}$$

$$\frac{\Delta \mid \Gamma, x \vdash e \qquad x \in (\Delta, \Gamma)}{\Delta \mid \Gamma \vdash \mathsf{dup}\ x;\ e}\ \textsc{dup} \qquad \frac{\Delta \mid \Gamma, \diamond_k \vdash e \qquad k \ = \ \mathsf{size}(x)}{\Delta \mid \Gamma, x \vdash \mathsf{dropru}\ x;\ e}\ \textsc{dropru}$$

$$\frac{\Delta \mid \Gamma, \diamond_k \vdash e \qquad k \geqslant 1}{\Delta \mid \Gamma \vdash \mathsf{alloc}\ k;\ e}\ \textsc{alloc} \qquad \frac{x \in (\Delta, \Gamma) \qquad \Delta \mid \Gamma, \overline{x}_i \vdash e_i}{\Delta \mid \Gamma \vdash \mathsf{match}\ x\ \{\ C_i\ \overline{x}_i \mapsto \mathsf{dup}\ \overline{x}_i;\ e_i\ \}}\ \textsc{match}$$

$$\frac{\Delta \mid \Gamma_1 \vdash e_1 \qquad \Delta \mid \Gamma_2 \vdash e_2}{\Delta \mid \Gamma_1, \Gamma_2 \vdash e_1\ e_2}\ \textsc{app} \qquad \frac{\varnothing \mid \overline{z}, \overline{x} \vdash e \qquad \overline{z} \ = \ \mathsf{fv}(\lambda\overline{x}.\ e)}{\Delta \mid \overline{z} \vdash \lambda^{\overline{z}}\,\overline{x}.\ e}\ \textsc{lam}$$

Fig. 8. The λ^{fip} calculus extends the Perceus linear resource calculus with borrowing, reuse, and unboxed tuples. The calculus extends the syntax, rules, and functional semantics of the FBIP calculus as shown in Figure 4 and 6. The multiplicity of each variable in Γ is unconstrained.

```
More(Pair(y, z, _), q, u) -> (More(Triple(x, y, z), q, u), BCons(b, Pad, Pad))
More(Triple(y, z, w), q, u) ->
  match! b
    BCons(b’, _, _) ->
      val (q’, b”) = cons(Pair(z, w, Pad), u3, q, b’)
      (More(Pair(x, y, Pad), q’, u), b”)
```

This function is now fully in-place. In the `More(Pair)` case we store an unneeded credit in the buffer. In the `More(Triple)` case we recurse[4] and take a reuse credit from the buffer. The buffer has the invariant that, given n_1 `Triple`, n_2 `Three` and n_3 `Two` constructors in the finger tree, its size is $n_1 \ + \ 2 * n_2 \ + \ n_3$. Since this invariant is maintained in all methods, the buffer is never empty.

5 REFERENCE COUNTING WITH BORROWING AND UNBOXING

In this section we formalize the connection between the FIP calculus and Perceus precise reference counting [Lorenzen and Leijen 2022; Reinking, Xie et al. 2021]. Our implementation of FIP in Koka uses this approach where detection of the uniqueness of owned arguments happens dynamically at run-time. Figure 8 formalizes the λ^{fip} calculus as an extension of the syntax and operational semantics of the FBIP calculus (in Figure 4 and 6). It has full lambda expressions $\lambda^{\overline{z}}\,\overline{x}.\ e$ now since arbitrary allocation is allowed. Here we write the free variables of the lambda expression explicitly as (the multiset) $\overline{z}$. This is not needed for the functional operational semantics but as we see later, we require it for the heap based operational semantics. Moreover, we have a $\mathsf{dup}\ x;\ e$ and $\mathsf{dropru}\ x;\ e$ expressions that let us duplicate owned variables and explicitly reuse dropped variables. Finally, the $\mathsf{alloc}\ k;\ e$ allows for abitrary allocation of constructors by creating reuse credits $\diamond_k$ at runtime.

The (beta) evaluation rule for lambda expressions is standard, and we can see that the (dup), (dropru), and (alloc) rules have no effect in the functional operational semantics (and are only used in the heap semantics).

We rephrase the original Perceus linear resource calculus, called λ^1 [Reinking, Xie et al. 2021], in Figure 8 as a *type system* (instead of a typed translation). We call any expressions in $\Delta \mid \Gamma \vdash e$

[4] The recursive call is here in tail-position modulo product-contexts [Leijen and Lorenzen 2023], which can be efficiently compiled to a tail-recursive call. However, we could also apply the TRMReC transformation to eliminate it (see App. D of the tech. report).

well-formed, and such expression always uses correct reference counting when evaluated, i.e. it never drops a value from the heap that is still needed later, or leaves garbage in the heap at the end of an evaluation. Moreover, the new type rules also extend the original rules with borrowing and unboxed tuples, and give a characterization of reuse based on reuse credits.

Just like the FIP calculus, the rules are still based on linear logic with a linear owned Γ environment, but unlike a pure linear logic it now has an escape hatch: through rules like dup we can freely duplicate "linear" variables by maintaining reference counts dynamically at runtime. As we can see, there is a suprisingly close connection between λ^{fip} and the FBIP and FIP calculi where each one is a strict subset of the other: FIP $\subset$ FBIP $\subset \lambda^{\mathsf{fip}}$. As such, the FIP calculus is exactly the subset of λ^{fip} that excludes the rules that require dynamic reference counting!

The dup rule is the rule that either allows us to use a borrowed variable ($x \in \Delta$) as owned, or duplicates an owned variable ($x \in \Gamma$). The alloc rule now allows arbitrary allocation of a constructor by adding a reuse credit $\diamond_k$ to the owned environment. With the full lambda expressions, we also have an app rule to apply an argument to a lambda expression. Here, we split the owned environment in two parts for each subexpression. We could have been more elaborate and allow borrowing of Γ_2 in the e_1 derivation just like our earlier let rule in Figure 4. We refrain from doing that here for simplicity as we can always use let if borrowing is required.

The lam rule requires that all free variables of the lambda expression are owned (which are needed to create the initial closure). In the body, we check with the free variables (from the closure) and the passed in parameters as all owned. Borrow information is not part of a type, so only top-level functions can take borrowed arguments (using the call rule).

The match rule can match on any borrowed or owned variable. However, each branch must start by dupping the matched constructor fields (as $\mathsf{dup}(\overline{x}_i)$). Indeed, since the match is non-destructive each field is now reachable directly but also via the original x (and thus we need to increment the reference count at runtime). For simplicity, the match rule can only match variables but we can always rewrite an expression $\mathsf{match}\ e\ \{\ \dots\ \}$ into $\mathsf{let}\ x\ =\ e\ \mathsf{in}\ \mathsf{match}\ x\ \{\ \dots\ \}$ for a fresh x when required. Since match no longer creates reuse credits, we can now create them explicitly instead using the "drop reuse" dropru rule. This drops a variable x, and immediately allows for a reuse credit $\diamond_k$ where k is the allocated size of x.

With the new match and dropru rules we no longer require the destructive match and corresponding rule of the FIP calculus and we can always replace any destructive match:

$$\mathsf{match!}\ x\ \{\ C_i\ \overline{x}_i \longrightarrow e_i\ \} \qquad \text{with} \qquad \mathsf{match}\ x\ \{\ C_i\ \overline{x}_i \longrightarrow \mathsf{dup}\ \overline{x}_i;\ \mathsf{dropru}\ x;\ e_i\ \}\ .$$

In particular, if the FIP match! expression is well-formed, we have:

$$\frac{\Delta \mid \Gamma, \overline{x}_i, \diamond_k \vdash\ e_i \ \ (\mathbf{1})}{\Delta \mid \Gamma, x\ \vdash\ \mathsf{match!}\ x\ \{\ C_i\ \overline{x}_i \longrightarrow e_i\ \}}\ \textsc{dmatch!}$$

and thus we can also derive that the translated match is well-formed in the λ^{fip} calculus:

$$\frac{x \in \Gamma, x \qquad \dfrac{\Delta \mid \Gamma, \overline{x}_i, \diamond_k \vdash\ e_i \ \ (1)}{\Delta \mid \Gamma, x, \overline{x}_i \vdash \mathsf{dropru}\ x;\ e_i}\ \textsc{dropru}}{\Delta \mid \Gamma, x \vdash\ \mathsf{match}\ x\ \{\ C_i\ \overline{x}_i \longrightarrow \mathsf{dup}\ \overline{x}_i;\ \mathsf{dropru}\ x;\ e_i\ \}}\ \textsc{match}$$

Furthermore, unlike the FIP or FBIP calculus, we can always elaborate a plain expression with dup, drop, free, alloc, and dropru to make it a well-formed λ^{fip} expression. The heap semantics can thus always be used to evaluate an expression. In particular, we can easily adapt the Perceus algorithm [Reinking, Xie et al. 2021] to elaborate plain expressions with correct reference count instructions.

A heap H extends a store S with reference counts $n \geqslant 1$, and can hold closures as well:

$$\mathsf{H} ::= \varnothing \mid \mathsf{H}, \diamond_k \mid \mathsf{H}, x \mapsto^n \varphi \qquad \text{where } \varphi ::= C\,\overline{x} \;\mid \lambda^{\overline{z}}\overline{x}.\ e$$

The $\longmapsto_{\mathsf{h}}$ relation extends the $\longmapsto_{\mathsf{s}}$ (using H for S) relation:

$$\begin{array}{llllll}
(dup_h) & \mathsf{H}, x \mapsto^n v & \mid \mathsf{dup}\ x;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, x \mapsto^{n+1} v & \mid e \\
(dlam_h) & \mathsf{H}, x \mapsto^1 \lambda^{\overline{z}}\overline{x}.e' & \mid \mathsf{drop}\ x;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H} & \mid \mathsf{drop}\ \overline{z};\ e \\
(drop_h) & \mathsf{H}, x \mapsto^{n+1} v & \mid \mathsf{drop}\ x;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, x \mapsto^n v & \mid e \qquad (\text{if } n \geqslant 1) \\
(dropru_h) & \mathsf{H}, x \mapsto^{n+1} C^k\,\overline{x} & \mid \mathsf{dropru}\ x;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, \diamond_k, x \mapsto^n C^k\,\overline{x} & \mid e \qquad (\text{if } n \geqslant 1) \\
(dconru_h) & \mathsf{H}, x \mapsto^1 C^k\,\overline{x} & \mid \mathsf{dropru}\ x;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, \diamond_k & \mid \mathsf{drop}\ \overline{x};\ e
\end{array}$$

$$\begin{array}{lllll}
(alloc_h) & \mathsf{H} & \mid \mathsf{alloc}\ k;\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, \diamond_k \mid e \\
(lam_h) & \mathsf{H} & \mid \lambda^{\overline{z}}\,\overline{x}.\ e & \longrightarrow_{\mathsf{h}} & \mathsf{H}, x \mapsto^1 \lambda^{\overline{z}}\overline{x}.\ e \mid x \qquad (\text{fresh } x) \\
(app_h) & \mathsf{H} & \mid (y)\ \overline{y} & \longrightarrow_{\mathsf{h}} & \mathsf{H} \mid \mathsf{dup}\ \overline{z};\ \mathsf{drop}\ y;\ e[\overline{x}{:=}\overline{y}] \quad (y \mapsto^n \lambda^{\overline{z}}\overline{x}.\ e \in \mathsf{H})
\end{array}$$

Fig. 9. Heap semantics of λ^{fip} – extending the FBIP store semantics as shown in Figure 5 and 6.

5.1 Heap Semantics

Figure 9 gives a heap based operational semantics for our Perceus calculus. Here we generalize the store S from the FIP calculus (Figure 5) to contain a reference count $n \geqslant 1$ for each binding. The heap now contains reuse credits $\diamond_k$, a constructor binding $x \mapsto^n C\,\overline{x}$, or a closure $x \mapsto^n \lambda^{\overline{z}}\,\overline{x}.\ e$. We extend the original FBIP store semantics in Figure 5 and Figure 6 with new rules where the evaluation context and EVAL rule stays the same (just replacing a store S with a heap H). For the ($bmatch$) rule we can allow any reference count on the matched binding, while the ($dmatch!$) rule requires that the matched binding has a unique reference count:

$$\begin{array}{lllll}
(bmatch_h) & \mathsf{H},\ y \mapsto^n C^k\,\overline{y} \mid \mathsf{match}\ y\ \{\overline{p \to e}\} & \longrightarrow_{\mathsf{h}} & \mathsf{H},\ y \mapsto^n C^k\,\overline{y} \mid e_i[\overline{x}{:=}\overline{y}] & (p_i\ =\ C^k\,\overline{y}) \\
(dmatch_h) & \mathsf{H},\ x \mapsto^1 C^k\,\overline{y} \mid \mathsf{match!}\ x\ \{\overline{p \to e}\} & \longrightarrow_{\mathsf{h}} & \mathsf{H}, \diamond_k \mid e_i[\overline{x}{:=}\overline{y}] & (p_i\ =\ C^k\,\overline{x})
\end{array}$$

The extra transition rules are for general allocation and reference counting. The ($alloc_h$) rule allows allocating a constructor without a reuse credit, and similarly, the (lam_h) rule allocates a closure. The application rule (app_h) applies a closure. We see that it starts by dupping its environment $\overline{z}$, and then dropping the closure itself. This way the APP rule can consider the free variables to part of the owned environment. This is important in practice as it allows a function to discard variables in the environment as soon as possible and be garbage-free. Here we can also see why we need to maintain $\overline{z}$ explicitly: even though the free variables of a lambda expression are initially distinct, during evaluation some may be substituted by the same variable and we need to dup such variable multiple times when applying to maintain proper reference counts.

The other rules all deal with reference counting. The (dup_h) transition increments a reference count, while ($drop_h$) decrements a reference count $n > 1$. The ($dlam_h$) rule drops a closure when the reference count is 1; this never creates a reuse credit though as the size of a closure cannot be accounted for statically. Note that also the environment is dropped, just like the ($dconru_h$) rule and the ($dcon_s$) rule in Figure 6. The ($dconru_h$) rule creates a reuse credit if the reference count is unique, while the ($dropru_h$) rule applies for a non-unique reference count with $n > 1$; this rule decrements the reference count but also allocates a fresh reuse credit (as required by rule DROPRU) – this is where the runtime falls back to copying if the cell was not unique.

Of course, with our match! translation, we no longer require the $(dmatch_h)$ rule and can derive the rule from the translated expression when we assume the matched binding is unique:

$$\begin{array}{lll}
 & \mathsf{H}, x \mapsto^1 C_i^k\,\overline{y} \mid \mathsf{match!}\ x\ \{\ C_i\,\overline{x}_i \to e_i\ \} & \\
= & \mathsf{H}, x \mapsto^1 C_i^k\,\overline{y} \mid \mathsf{match}\ x\ \{\ C_i\,\overline{x}_i \to \mathsf{dup}\ \overline{x}_i;\ \mathsf{dropru}\ x;\ e_i\ \} & \\
\longrightarrow_{\mathsf{h}} & \mathsf{H}, x \mapsto^1 C_i^k\,\overline{y} \mid \mathsf{dup}\ \overline{y};\ \mathsf{dropru}\ x;\ e_i[\overline{x}_i{:=}\overline{y}] & \\
\longrightarrow_{\mathsf{h}} & \mathsf{H}', x \mapsto^1 C_i^k\,\overline{y} \mid \mathsf{dropru}\ x;\ e_i[\overline{x}_i{:=}\overline{y}] & \{\ \mathsf{H}'\ \textit{is}\ \mathsf{H}\ \textit{but with all}\ \overline{y}\ \textit{refcounts} +1\ (\mathbf{A})\ \} \\
\longrightarrow_{\mathsf{h}} & \mathsf{H}', \diamond_k \mid \mathsf{drop}\ \overline{y};\ e_i[\overline{x}_i{:=}\overline{y}] & \{\ (dconru_h),\ rc\ \textit{is}\ 1\ \} \\
\longrightarrow_{\mathsf{h}} & \mathsf{H}, \diamond_k \mid e_i[\overline{x}_i{:=}\overline{y}] & \{\ (drop_h),\ (A)\ \}
\end{array}$$

However, the translation is more general, and can also proceed if the matched binding is not unique but shared – in that case the final steps use $(dropru_h)$ and become:

$$\begin{array}{lll}
 & \ldots & \\
\longrightarrow_{\mathsf{h}} & \mathsf{H}', x \mapsto^{n+1} C_i^k\,\overline{y} \mid \mathsf{dropru}\ x;\ e_i[\overline{x}_i{:=}\overline{y}] & \{\ \mathsf{H}'\ \textit{is}\ \mathsf{H}\ \textit{but with all}\ \overline{y}\ \textit{refcounts} +1\ (\mathbf{A})\ \} \\
\longrightarrow_{\mathsf{h}} & \mathsf{H}', \diamond_k, x \mapsto^{n} C_i^k\,\overline{y} \mid e_i[\overline{x}_i{:=}\overline{y}] & \{\ (dropru_h)\ \}
\end{array}$$

where the binding for x stays alive but we still allocate a fresh reuse credit. This is exactly where we can generate the code shown in Section 1.4 where we essentially inline and specialize the definition of dropru and check upfront if the matched binding is unique or not.

5.2 Soundness of the Heap Semantics

First we generalize the properties of the store semantics to reference counted heaps:

Definition 3. (*Heap Soundness and Linearity*)
For a heap H we write $\mathsf{dom}(\mathsf{H})$ to denote the set of variables x bound in H and write $\mathsf{rng}(\mathsf{H})$ to denote the set of values φ bound in H. Two heaps H_1, H_2 are compatible if they map equal names $x \mapsto^n v\ \in H_1,\ x \mapsto^m w\ \in H_2$, to equal values $v\ =\ w$. A heap is *sound* if all free variables in $\mathsf{rng}(\mathsf{H})$ are bound: $\mathsf{fv}(\mathsf{rng}(\mathsf{H})) \subseteq \mathsf{dom}(\mathsf{H})$. A heap is *linear* if it is sound, and any variable $x \mapsto^n v$ in $\mathsf{dom}(\mathsf{H})$ occurs at most n times in the free variables of $\mathsf{rng}(\mathsf{H})$. By $\mathsf{roots}(\mathsf{H})$ we denote the multi-set of reuse credits of H and variables $x \mapsto^n v$ of $\mathsf{dom}(\mathsf{H})$, which contains any variable $n\ -\ m$ times, if it occurs m times in the free variables of $\mathsf{rng}(\mathsf{H})$.

The definition of linearity ensures that mutation is safe if the reference count is one. Exactly as in the store semantics, we write $[\mathsf{H}]\overline{x}$ to denote a substitution that recursively replaces variables by their bound value in H. We assume that we are given heaps corresponding to the owned and borrowed values, but only require that the heap of the owned values is linear and do not assume that the heaps have a disjoint domain. Instead we use the *join* operator $\otimes$ to define a joined heap of the borrowed and owned part, even if they have common elements with the same name and value. We join common elements by summing their reference counts. Since this eliminates one reference to their children, we decrease their reference count accordingly:

$$\begin{array}{lcll}
\varnothing \otimes H_2 & = & H_2 & \\
H_1, \diamond_k \otimes H_2 & = & H_1 \otimes H_2, \diamond_k & \\
H_1, x \mapsto^n v \otimes H_2 & = & H_1 \otimes H_2, x \mapsto^n v & \text{iff } x \notin \mathsf{dom}(H_2) \\
H_1, x \mapsto^n v \otimes H_2, x \mapsto^m v, \overline{z \mapsto^{k+1} w} & = & H_1 \otimes H_2, x \mapsto^{n+m} v, \overline{z \mapsto^{k} w} & \text{iff } \overline{z}\ =\ \mathsf{fv}(v)
\end{array}$$

The rootset of $H_1 \otimes H_2$ is exactly the disjoint union of the roots of H_1 and H_2. When applied to linear heaps, $\otimes$ can be viewed as a partial commutative monoid [Jensen and Birkedal 2012] where every result is valid. However, we prefer a categorical view on linear heaps where a morphism $H_1 \to H_2$ exists if $\mathsf{dom}(H_1)\ \subseteq \mathsf{dom}(H_2)$ and $\mathsf{roots}(H_1)\ \subseteq \mathsf{roots}(H_2)$. This forms a monoidal category with the join operator as tensor product. We also define a heap subtraction operation $[H_1, H_2]$ if $H_1 \to H_2$ similar to an internal hom (and which corresponds to the magic wand of separation logic). We can

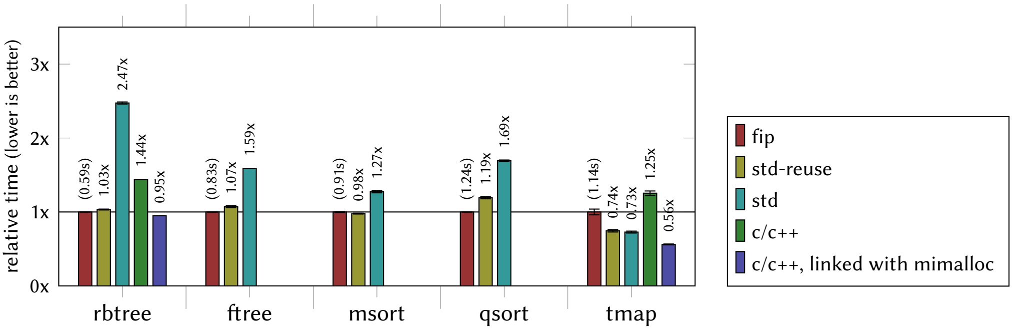

Fig. 10. Benchmarks on Ubuntu 22.04.2 (AMD 7950x), Koka v2.4.2.

then show that heap evaluation leaves the borrowed values unchanged:

Theorem 6. (*The heap semantics is sound for well-formed Perceus programs*)
If $\Delta \mid \Gamma \vdash e$ and given heaps H_1, H_2 with $\Delta \subseteq \mathrm{dom}(H_1)$, H_1 sound, $\Gamma = \mathrm{roots}(H_2)$ and H_2 linear, then $[H_1 \otimes H_2]e \longmapsto^* \overline{v}$ implies $H_1 \otimes H_2 \mid e \longmapsto^*_{\mathsf{h}} H_1 \otimes H_3 \mid \overline{x}$ where $[H_3]\overline{x} = \overline{v}$, $\overline{x} = \mathrm{roots}(H_3)$ and H_3 is linear.

This is again a strong theorem as it shows that the dynamic reference count is always correct and no variables will be discarded too early, while also having no garbage at the end of an evaluation ($\overline{x} = \mathrm{roots}(H_3)$). Our proof (in App. E of the tech. report) is novel and may be well suited to possible mechanized formalization. As a corollary, any closed λ^{fip} expression can evaluate starting from an empty heap:

Corollary 2.
If $e \longmapsto^* \overline{v}$ and $\varnothing \mid \varnothing \vdash e$, then $\varnothing \mid e \longmapsto^*_{\mathsf{h}} \mathsf{H} \mid \overline{x}$ and $[\mathsf{H}]\overline{x} = \overline{v}$.

While it is outside the scope of this paper, we could also modify the LET rule of our calculus with a ($\star$)-condition to characterize *garbage-free* and *frame-limited* derivations [Lorenzen and Leijen 2022]. However, borrowing makes it harder to achieve these proporties and further study is needed. In particular, a garbage-free derivation can only exist if all borrowed arguments are still used later on, and similarly, a frame-limited derivation can only exist if all borrowed arguments are either used later on or have constant size.

6 BENCHMARKS

Figure 10 shows benchmark results of examples from this paper, relative to the *fip* variant. The results are the average over 5 runs on an AMD7950X on Ubuntu 22.04.2 with Koka v2.4.2. Each benchmark uses 100 iterations over N (=100000) element structures. We test each benchmark in following variants:

- *fip*: the algorithm implemented as FIP in Koka.
- *std*: the standard functional implementation in Koka without reuse optimization. For general GC'd languages without precise reference counts, the relative performance between *std* and *fip* can be more indicative of potential performance gains.
- *std-reuse*: just as *std* but with reuse optimization enabled. This is standard Koka which always applies dynamic reuse.
- *c/c++*: an standard in-place updating implementation in C or C++. Since our benchmarks are allocation heavy, we also include a variant when linked with the `mimalloc` [Leijen et al. 2019]

memory allocator since that is usually faster than the standard C/C++ one.

The benchmarks consist of:

- *rbtree*: performs N balanced red-black tree insertions and folds the tree to compute the sum of the elements. The *fip* variant is adapted from Lorenzen and Leijen [2022]. while *std* uses Okasaki style insertion [Okasaki 1999]. The C++ versions use the standard in-place updating STL `std::map` which is implemented using red-black trees internally.
- *ftree*: builds a finger tree of size N and performs 3*N uncons/snoc operations. The *fip* variant is shown in Section 4.3 where the *std* variant uses a implementation described by Claessen [2020].
- *msort*, *qsort*: sorts an N element random list. The *fip* variants use the implementations shown in Section 4.2 while *std* uses the standard recursive functional implementations (derived from the Haskell library implementations).
- *tmap*: maps an increment function over a *shared* (non-unique) N element tree returning a fresh tree which is then folded to compute the sum of the elements. The *fip* variant uses the implementation of Section 3 while *std* and *c/c++* use the standard (recursive) way to map over a tree.

It is hard to draw firm conclusions as the results are dependent on our particular implementation, but we make some general observations:

- The performance of *fip* versus *std* is generally much better showing that in-place updating is indeed generally faster than allocation.
- Even without a `fip` annotation, the *std-reuse* variant shows that the reuse optimization in Koka can be very effective – but of course, unlike `fip`, reuse here is not guaranteed.
- In an absolute sense, the performance seems very good where in the *rbtree* benchmark the *fip* variant rivals the performance of the in-place updating `std::map` implementation in C++.
- The *tmap* benchmark is interesting as *fip* is generally slower here. The *fip* variant uses a zipper to visit the tree and uses constant stack space (unlike the others which use stack space linear in the depth of the tree). Reversing the pointers Schorr-Waite style can be slower though than recursing with the stack. Also, the tree that is mapped is shared and thus even the `fip` function cannot reuse the original tree. Nevertheless, the *fip* variant will still reuse the zipper it uses to traverse the tree. This is also shows why *std* and *std-reuse* are performing similarly, since there is no reuse possible for the standard algorithms. That *std-reuse* is only about 1% slower shows that the dynamic reuse check has negligible impact on performance.

7 RELATED WORK

The FIP calculus is most closely related to Hofmann's type system for in-place update [Hofmann 2000b 2000a]. Just like Hofmann, we add reuse credits to a linear enviroment, model a destructive match, and collect top-level functions in the signature. However, Hofmann's unboxed tuples can escape into allocations, which makes it necessary to monomorphise the program (and track types to be able to do so). In contrast, our calculus does not need monomorphisation or know about types at all. Hofmann also uses a uniform size for all constructors of a datatype (including atoms such as `Nil`), but unboxes the first layer of each datatype. Many FIP programs can also be checked by that scheme, but it seems to increase memory usage substantially: in our calculus, a constructor with n fields filled with atoms takes n space, while it would take $n * n$ space in Hofmann's calculus.

While we only model unique and borrowed values in our FIP calculus, *shared* values are another interesting variant. Unlike borrowed values, shared values can be stored in datatypes. But unlike unique values, they can be used multiple times (and it is not possible to use a destructive `match!` on them). Shared values correspond to the usage aspect 2 introduced by Aspinall and Hofmann [2002] and Aspinall et al. [2008]. We believe that it may be worthwhile to extend the FIP calculus with shared values to allow it to check a wider range of programs. However, shared values can only be supported in a garbage-collected setting, while our FIP programs can also easily be compiled to C.

Even without in-place reuse, FIP programs still use constant space, which allows us to reason about their space usage. Space credits [Hofmann 2003; Hofmann and Jost 2003] generalize reuse credits with the axiom $\diamond_{n_1}, \diamond_{n_2} \;=\; \diamond_{n_1 + n_2}$. This axiom does not hold for reuse credits (which can not be combined unless they are in adjacent slots in the heap), but it does hold if we view $\diamond_n$ just as the promise that n words of space is available. Based on space credits, an automated analysis [Hoffmann et al. 2011; Hofmann and Jost 2006] or manual proofs in separation logic [Madiot and Pottier 2022; Moine et al. 2023] can be used to reason about heap space. However, these systems usually do not model atoms or unboxing, which we identified as crucial for real-world FIP programs.

Reuse analysis can be implemented either statically using uniqueness types [Barendsen and Smetsers 1995] or flow analysis, or dynamically using reference counts. Compile-time Garbage Collection [Bruynooghe 1986] is the most developed flow based analysis, which tracks the flows of unique values through the program to identify reuse opportunities statically. Reuse with reference counts has long been applied to arrays, where the update function can be designed to mutate the array in-place if the reference count is one [Hudak and Bloss 1985; Scholz 1994]. Similarly, Stoye et al. [1984] used reuse with one-bit reference counts for combinator reduction, where reuse is encoded in the hand-written combinators. However, in this work, we rely on a reuse analysis that can statically discover reuse opportunities between otherwise unconnected memory cells, which was pioneered by OPAL [Didrich et al. 1994; Schulte 1994; Schulte and Grieskamp 1992]. Their analysis was refined by Ullrich and de Moura [2019], who showed that such an analysis can be implemented efficiently without duplicating code. Reinking, Xie et al. [2021] present the linear resource calculus as a formalization of precise reference counting and give a *garbage free* algorithm. Lorenzen and Leijen [2022] refine this calculus further with a declarative star condition that can guarantee either garbage-free or *frame-limited* space usage which ensures extra space usage due to reuse is bounded.

Borrowing is a long-standing technique to reduce the overhead of reference counting [Baker 1994b; Lemaitre et al. 1986]. If a lifetime analysis can prove that the lifetime of one reference dominates that of another, we can avoid counting the second reference – in our calculus, this is expressed by borrowing Γ_2 in LET (first introduced by [Reinking, Xie et al. 2021]). Ullrich and de Moura [2019] introduced borrowed parameters on top-level functions which is especially important for recursive functional code. They also showed an inference for borrowing annotations, but, as pointed out by Lorenzen and Leijen [2022], this can increase memory usage by an unbounded amount.

8 CONCLUSION AND FUTURE WORK

We have shown the necessary features for, and spirit of, fully in-place functional programming. We believe the examples given in this paper have only scratched the surface of what is possible and that there are more FIP algorithms waiting to be discovered. Another interesting research direction is to extend the fully in-place calculus to cover more possible programs that are currently not quite FIP; for example by adding shared values as described by Aspinall et al. [2008].

REFERENCES

David Aspinall, and Martin Hofmann. 2002. Another Type System for in-Place Update. In *ESOP*, 2:36–52. Springer. doi:https://doi.org/10.1007/3-540-45927-8_4.

David Aspinall, Martin Hofmann, and Michal Konečný 2008. A Type System with Usage Aspects. *Journal of Functional Programming* 18 (2). Cambridge University Press: 141–178. doi:https://doi.org/10.1017/S0956796807006399.

Roland C Backhouse. 1988. *An Exploration of the Bird-Meertens Formalism.* University of Groningen, Department of Mathematics and Computing Science.

Henry G Baker. 1994a. A "linear Logic" Quicksort. *ACM Sigplan Notices* 29 (2). ACM New York, NY, USA: 13–18. doi:https://doi.org/10.1145/181748.181750.

Henry G Baker. 1994b. Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures. *ACM Sigplan Notices* 29 (9). ACM New York, NY, USA: 38–43. doi:https://doi.org/10.1145/185009.185016.

Erik Barendsen, and Sjaak Smetsers. 1995. Uniqueness Type Inference. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, 189–206. doi:https://doi.org/10.1007/BFb0026821.

Frédéric Bour, Basile Clément, and Gabriel Scherer. Apr. 2021. Tail Modulo Cons. *Journeées Francophones Des Langages Applicatifs (JFLA)*, April. Saint Médard d'Excideuil, France. doi:https://doi.org/10.48550/arXiv.2102.09823. hal-03146495.

Tom H Brus, Marko CJD van Eekelen, MO Van Leer, and Marinus J Plasmeijer. 1987. Clean—a Language for Functional Graph Rewriting. In *Functional Programming Languages and Computer Architecture: Portland, Oregon, USA, September 14–16, 1987 Proceedings*, 364–384. Springer. doi:https://doi.org/10.1007/3-540-18317-5_20.

Maurice Bruynooghe. 1986. *Compile Time Garbage Collection*. Katholieke Universiteit Leuven. Departement Computerwetenschappen.

Koen Claessen. 2020. Finger Trees Explained Anew, and Slightly Simplified (functional Pearl). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, 31–38. doi:https://doi.org/10.1145/3406088.3409026.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. MIT press.

Olivier Danvy. 2008. From Reduction-Based to Reduction-Free Normalization. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, 66–164. AFP'08. Springer-Verlag, Berlin, Heidelberg. doi:https://doi.org/10.1007/978-3-642-04652-0_3.

Olivier Danvy. 2022. Getting There and Back Again. *Fundamenta Informaticae* 185. Episciences. org. doi:https://doi.org/10.3233/FI-222106.

Edsko De Vries, Rinus Plasmeijer, and David M Abrahamson. 2008. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers 19*, 201–218. Springer. doi:https://doi.org/10.1007/978-3-540-85373-2_12.

Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. 1994. OPAL: Design and Implementation of an Algebraic Programming Language. In *Programming Languages and System Architectures*, 228–244. Springer. doi:https://doi.org/10.1007/3-540-57840-4_34.

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. doi:https://doi.org/10.1145/155090.155113.

Jeremy Gibbons. 1994. An Introduction to the Bird- Meertens Formalism.

J.R. Hindley. Dec. 1969. The Principal Type Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society* 146 (December): 29–60. doi:https://doi.org/10.2307/1995158.

Ralf Hinze, and Ross Paterson. 2006. Finger Trees: A Simple General-Purpose Data Structure. *Journal of Functional Programming* 16 (2). Cambridge University Press: 197–217. doi:https://doi.org/10.1017/S0956796805005769.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 357–370. doi:https://doi.org/10.1145/1926385.1926427.

Martin Hofmann. 2000a. In-Place Update with Linear Types or How to Compile Functional Programms into Malloc-Free C. *Preprint,* . Citeseer.

Martin Hofmann. 2000b. A Type System for Bounded Space and Functional in-Place Update. In *European Symposium on Programming*, 165–179. Springer. doi:https://doi.org/10.1007/3-540-46425-5_11.

Martin Hofmann. 2003. Linear Types and Non-Size-Increasing Polynomial Time Computation. *Information and Computation* 183 (1). Elsevier: 57–85. doi:https://doi.org/10.1016/S0890-5401(03)00009-9.

Martin Hofmann, and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. *ACM SIGPLAN Notices* 38 (1). ACM New York, NY, USA: 185–197. doi:https://doi.org/10.1145/604131.604148.

Martin Hofmann, and Steffen Jost. 2006. Type-Based Amortised Heap-Space Analysis. In *European Symposium on Programming*, 22–37. Springer. doi:https://doi.org/10.1007/11693024_3.

Paul Hudak. 1986. A Semantic Model of Reference Counting and Its Abstraction (detailed Summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 351–363. doi:https://doi.org/10.1145/319838.319876.

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. doi:https://doi.org/10.1145/318593.318660.

Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7 (5). Cambridge University Press: 549–554. doi:https://doi.org/10.1017/s0956796897002864.

R John Muir Hughes. 1986. A Novel Representation of Lists and Its Application to the Function "reverse." *Information Processing Letters* 22 (3). Elsevier: 141–144. doi:https://doi.org/10.1016/0020-0190(86)90059-1.

Jonas Braband Jensen, and Lars Birkedal. 2012. Fictional Separation Logic. In *Programming Languages and Systems: 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24-April 1, 2012. Proceedings 21*, 377–396. Springer. doi:https://doi.org/10.1007/978-3-642-28869-2_19.

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

Daan Leijen, and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proceedings of the ACM on Programming Languages* 7 (POPL). ACM New York, NY, USA: 1152–1181. doi:https://doi.org/10.1145/3571233.

Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action. In *Asian Symposium on Programming Languages and Systems*, 244–265. Springer. doi:https://doi.org/10.1007/978-3-030-34175-6_13.

Michel Lemaitre, Michel Castan, M-H Durand, Guy Durrieu, and Bernard Lecussan. 1986. Mechanisms for Efficient Multiprocessor Combinator Reduction. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 113–121. doi:https://doi.org/10.1145/319838.319855.

Alexey Loginov, Thomas Reps, and Mooly Sagiv. 2006. Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In *Static Analysis: 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006. Proceedings 13*, 261–279. Springer. doi:https://doi.org/10.1007/11823230_17.

Anton Lorenzen, and Daan Leijen. 2022. Reference Counting with Frame Limited Reuse. *Proceedings of the ACM on Programming Languages* 6 (ICFP). ACM New York, NY, USA: 357–380. doi:https://doi.org/10.1145/3547634.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. May 2023a. *FP^2: Fully in-Place Functional Programming Artifact.* Zenodo. doi:https://doi.org/10.5281/zenodo.7988150. Artifact for ICFP'23.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. May 2023b. *FP^2: Fully in-Place Functional Programming.* MSR-TR-2023-19. Microsoft Research.

Jean-Marie Madiot, and François Pottier. 2022. A Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 6 (POPL). ACM New York, NY, USA: 1–28. doi:https://doi.org/10.1145/3498672.

Conor McBride. 2001. The Derivative of a Regular Type Is Its Type of One-Hole Contexts. *Unpublished Manuscript*, 74–88.

Conor McBride. 2008. Clowns to the Left of Me, Jokers to the Right (pearl) Dissecting Data Structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 287–295. doi:https://doi.org/10.1145/1328438.1328474.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17: 248–375. doi:https://doi.org/10.1016/0022-0000(78)90014-4.

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 7 (POPL). ACM New York, NY, USA: 718–747. doi:https://doi.org/10.1145/3571218.

Chris Okasaki. 1999. Red-Black Trees in a Functional Setting. *Journal of Functional Programming* 9 (4). Cambridge University Press: 471–477. doi:https://doi.org/10.1017/s0956796899003494.

Simon L. Peyton Jones, and John Launchbury. 1991. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *Functional Programming Languages and Computer Architecture*, edited by John Hughes, 636–666. Springer Berlin Heidelberg. doi:https://doi.org/10.1007/3540543961_30.

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. Virtual, Canada. doi:https://doi.org/10.1145/3453483.3454032.

John C Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference-Volume 2*, 717–740. doi:https://doi.org/10.1145/800194.805852.

John C Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 55–74. IEEE. doi:https://doi.org/10.1109/LICS.2002.1029817.

Sven-Bodo Scholz. 1994. Single Assignment C-Functional Programming Using Imperative Style. In *Proceedings of IFL*, volume 94.

Herbert Schorr, and William M Waite. 1967. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM* 10 (8). ACM New York, NY, USA: 501–506. doi:https://doi.org/10.1145/363534.363554.

Wolfram Schulte. 1994. Deriving Residual Reference Count Garbage Collectors. In *International Symposium on Programming Language Implementation and Logic Programming*, 102–116. Springer. doi:https://doi.org/10.1007/3-540-58402-1_9.

Wolfram Schulte, and Wolfgang Grieskamp. 1992. Generating Efficient Portable Code for a Strict Applicative Language. In *Declarative Programming, Sasbachwalden 1991*, 239–252. Springer. doi:https://doi.org/10.1007/978-1-4471-3794-8_16.

Daniel Dominic Sleator, and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *Journal of the ACM* 32: 652–686. doi:https://doi.org/10.1145/3828.3835.

Jonathan Sobel, and Daniel P Friedman. 1998. Recycling Continuations. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, 251–260. doi:https://doi.org/10.1145/289423.289452.

William R Stoye, Thomas JW Clarke, and Arthur C Norman. 1984. Some Practical Methods for Rapid Combinator Reduction. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 159–166. doi:https://doi.org/10.1145/800055.802032.

Sebastian Ullrich, and Leonardo de Moura. 2019. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, 1–12. doi:https://doi.org/10.1145/3412932.3412935.

Twan van Laarhoven. 2007. Deriving Functor. `https://mail.haskell.org/pipermail/haskell-prime/2007-March/002137.html`.

David Walker, and Greg Morrisett. 2000. Alias Types for Recursive Data Structures. *Types in Compilation* 2071. Springer: 177–206. doi:https://doi.org/10.1007/3-540-45332-6_7.

Hongseok Yang. 2007. Relational Separation Logic. *Theoretical Computer Science* 375 (1-3). Elsevier: 308–334. doi:https://doi.org/10.1016/j.tcs.2006.12.036.