FP²: Fully in-Place Functional Programming

Technical Report MSR-TR-2023-19, 2023-07-03, v2

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.

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

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]:

```haskell
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 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.
FP²: Fully in-Place Functional Programming

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 integers¹:

```
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 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! keyword² indicates a destructive match, after which the matched variable t can no longer be used. Intuitively, we can then see straightforwardly that the matched Node cell has become redundant and can be reused.

¹All examples in this paper are written in the Koka language which has a full implementation of the FIP check (v2.4.2).
²In 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.
for the \texttt{NodeL} cell of the zipper.

To make this intuition more precise, we view a destructively matched constructor, such as \texttt{Node(1,x,r)}, as a collection of its children \texttt{l}, \texttt{x}, and \texttt{r}, and a \texttt{reuse credit} \texttt{⋄}. This “diamond” resource type \texttt{⋄_k} represents a specific heap cell of size \texttt{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 \texttt{match!} introduces a reuse credit \texttt{⋄_3} for the \texttt{Node}, which is consumed by the allocation of \texttt{NodeL} which allows our \texttt{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 \texttt{unboxed} values hence no allocation is needed for these\footnote{In Koka, tuples are implemented as \texttt{value types} which are unboxed and passed in registers. The \texttt{fip} keyword additionally checks that no automatic (heap allocated) boxing is applied for such value types inside a FIP function.}.

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

\begin{verbatim}
fip fun reverse-acc( xs : list\langle a\rangle, acc : list\langle a\rangle ) : list\langle a\rangle
match! xs
  Cons(x,xx) -> reverse-acc( xx, Cons(x,acc) )
  Nil -> acc
\end{verbatim}

where the destructive match on \texttt{xs} allows the matched \texttt{Cons} cell to be reused (\texttt{⋄_2}) for the \texttt{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 \texttt{lookup} function as follows:

\begin{verbatim}
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
\end{verbatim}

The \texttt{lookup} function calls \texttt{zlookup} with an initial empty context \texttt{Root}. It seems we need to allocate the \texttt{Root} constructor, but constructors without any fields do not require allocation. These are typically implemented using pointer-tagging where only the tag is used to represent them. We call such constructors \texttt{atoms}. Examples include the \texttt{Nil} of lists, but also booleans (\texttt{True} and \texttt{False}), and, depending on the implementation, primitive types like integers (\texttt{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:

```plaintext
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:

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

```plaintext
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:

```ocaml
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:

```ocaml
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 transformed 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:

```ocaml
fun palindrome( xs : list<> ) : list<>
  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 \texttt{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 \texttt{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 \texttt{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:

\begin{verbatim}
\texttt{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}
\end{verbatim}

The reuse credit $\diamond_2$ is compiled into an explicitly named reuse token \texttt{ru}, and holds to the memory location of the resulting \texttt{Cons} cell. If the input list is unique, we reuse the address of the input, \texttt{&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 \texttt{Cons} cell in-place at the \texttt{ru} memory location as \texttt{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 \texttt{Cons} nodes for the resulting list.

We show formally that the FIP calculus can be embedded, as a subcalculus, into a Perceus-style linear resource calculus $\lambda_{fip}$ with unboxing, atoms and borrowing in Section 5.

This language can now express the full range of functional programming. Yet, only one rule needs to be extended: rather than always reusing in a destructive \texttt{match!}, we inspect the reference counts and only reuse unique cells as above. In particular, this shows that fully in-place functions can execute on unique arguments with no overhead due to reference counting (besides the \texttt{is-unique} check).

In this setting, when we call a function like \texttt{map} fully in-place, it means that the function itself does not intrinsically allocate any memory. However, any particular evaluation can still allocate by passing in a non-fip function, for example \texttt{map(xs, fn(x) Cons(x,Nil))}. That is an essential part
of our approach though, where we are able to reason about a fully in-place function like \texttt{map} in isolation but still allow it to be used in allocating scenarios from a non-FIP context.

In the next section we look at a precise definition of a fully in-place calculus.

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, $\lambda^l$ [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 ($\lambda^\text{fip}$) which includes all these features, and give a novel proof of soundness. Surprisingly, it turns out that the extended linear resource calculus $\lambda^\text{fip}$ can be seen a pure extension of the FIP calculus; and the rules of the FIP calculus are a strict subset of $\lambda^\text{fip}$. In particular, FIP is exactly that subset of $\lambda^\text{fip}$ 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 $\text{fip}$ 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, $\text{fip}$ functions tend to outperform with a larger margin.

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 let $\overline{x} = e_1$ 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 $\Sigma$ 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 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 $\square$ in place of a

---

**Figure 2. Syntax of the FIP calculus.**

**Fig. 2. Syntax of the FIP calculus.**
Evaluation order:

\[
E := \square | E \mid \bar{v} E \mid f(E; e) \mid f(\bar{v}; E) \mid \text{let } \bar{x} = E \text{ in e} \quad \begin{array}{c}
E[1] \rightarrow E[2] \quad \text{STEP}
\end{array}
\]

Evaluation steps:

\[
\begin{array}{lll}
\text{match } E \{ p \mapsto \bar{e} \} \quad \text{match! } E \{ p \mapsto \bar{e} \} & \quad & \begin{array}{c}
e_1 \rightarrow e_2
\end{array}
\end{array}
\]

\[
\begin{array}{lll}
\text{(let)} & \quad \text{let } \bar{x} = \bar{v} \text{ in } e & \quad \rightarrow e[\bar{x}:=\bar{v}]
\end{array}
\]

\[
\begin{array}{lll}
\text{(call)} & \quad f(\bar{v}_1; \bar{v}_2) & \quad \rightarrow e[\bar{y}:=\bar{v}_1, \bar{x}:=\bar{v}_2] \quad \text{with } f(\bar{y}, \bar{x}) = e \in \Sigma
\end{array}
\]

\[
\begin{array}{lll}
\text{(app)} & \quad (f) \bar{v} & \quad \rightarrow e[\bar{x}:=\bar{v}] \quad \text{with } f(; \bar{x}) = e \in \Sigma
\end{array}
\]

\[
\begin{array}{lll}
\text{(match)} & \quad \text{match } (C \bar{v}) \{ p \mapsto \bar{e} \} & \quad \rightarrow e[\bar{y}:=\bar{v}] \quad \text{with } p_i = C \bar{v}
\end{array}
\]

\[
\begin{array}{lll}
\text{(match!)} & \quad \text{match! } (C \bar{v}) \{ p \mapsto \bar{e} \} & \quad \rightarrow e[\bar{x}:=\bar{v}] \quad \text{with } p_i = C \bar{x}
\end{array}
\]

Fig. 3. Functional operational semantics.

sub term. For example, if \( E = f(\bar{v}; \square) \), then \( E[(x, y)] \) is \( f(\bar{v}; (x, y)) \). 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 \( e_1 \rightarrow e_2 \) evaluates \( e_1 \) to \( e_2 \). The reduction steps are standard except for always using unboxed tuples to substitute. We write \( e[\bar{x}:=\bar{v}] \) for the capture-avoiding substitution of the distinct variables \( \bar{x} = (x_1, \ldots, x_n) \) with the values \( \bar{v} = (v_1, \ldots, v_m) \), where \( n \) must be equal to \( m \) and \( \bar{x} \notin \text{fv}(\bar{v}) \). When we substitute in a function body, we write \( e[\bar{x}:=\bar{v}, \bar{y}:=\bar{v}_2] \) to substitute all (distinct) variable \( \bar{x} \) and \( \bar{y} \) at once, where we again require a capture avoiding substitution with \( \bar{x}, \bar{y} \notin \text{fv}(\bar{v}_1) \cup \text{fv}(\bar{v}_2) \).

When an expression \( e \) cannot be reduced further using step, then either \( e \) reduced to an unboxed tuple \( \bar{v} \) 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 \( \Delta \) and owned environment \( \Gamma \), the expression \( e \) is a well-formed FIP expression. The borrowed environment \( \Delta \) is a set of borrowed variables which generally come from the borrowed parameters of a function \( f \), or by borrowing in the left rule. We write \( \Delta, \Delta' \) for the union of the sets \( \Delta \) and \( \Delta' \). The owned environment \( \Gamma \) is a multiset of owned variables, and also reuse credits. Following Hofmann [2000a] we denoted these as a diamond \( \odot_k \), signifying a credit of size \( k \). We can append two owned environments \( \Gamma \) and \( \Gamma' \) as \( \Gamma, \Gamma' \). Sometimes, we also write \( \Delta, \Gamma \) to join a borrowed set with a multi-set \( \Gamma \) where it is required that there are no reuse credits in \( \Gamma \), where the result is the borrowed set \( \Delta \) joined with the elements in \( \Gamma \). Note that in the current rules, all variables in the \( \Gamma \) environment occur only once as we have no way to duplicate them. In Section 5 we generalize this to the full Perceus calculus.

The FIP rules ensure that variables in the owned environment \( \Gamma \) are used linearly (with some borrowing allowed in left). However, this is a syntaxic 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 \( \Gamma \). Similarly, the tuple rule splits the owned environment in \( n \) distinct parts, \( \Gamma_i \), and ensures well-formedness of each constituent value of the tuple, \( v_i \), under the corresponding environment \( \Gamma_i \). With the atom rule, \( \Delta \mid \emptyset \vdash C \) we can return
FP$^2$: Fully in-Place Functional Programming

\[
\begin{align*}
\Gamma & ::= \emptyset | \Gamma, x | \Gamma, \varnothing \kappa & \text{(owned environment)} \\
\Delta & ::= \emptyset | \Delta, y & \text{(borrowed environment)}
\end{align*}
\]

\[
\begin{array}{ll}
\frac{}{\Delta \mid x \vdash x} & \text{VAR} \\
\frac{\Delta \mid \Gamma_1 \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash (v_1, \ldots, v_n)} & \text{TUPLE} \\
\frac{\overline{y} \in \Delta, \text{dom}(\Sigma) \quad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash f(\overline{y}; e)} & \text{CALL} \\
\frac{y \in \Delta \quad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash y \ e} & \text{BAPP} \\
\frac{\Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma, \varnothing \kappa \vdash e} & \text{EMPTY} \\
\frac{\Gamma \vdash \emptyset}{\emptyset \vdash \emptyset} & \text{DEFBASE}
\end{array}
\]

\[
\begin{array}{ll}
\frac{}{\Delta \mid \emptyset \vdash C} & \text{ATOM} \\
\frac{\Delta \mid \Gamma \vdash v_i}{\Delta \mid \Gamma \vdash v_i} & \text{REUSE} \\
\frac{\Delta \mid \Gamma_1 \vdash v_i}{\Delta \mid \Gamma_1 \vdash v_i} & \text{REUSE} \\
\frac{\Delta, \Gamma_1 \mid \Gamma_2 \vdash e_1 \quad \Delta, \Gamma_2 \vdash \overline{x} \vdash e_2 \quad \overline{x} \notin \Delta, \Gamma_2, \Gamma_3}{\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3 \vdash \text{let } \overline{x} = e_1 \text{ in } e_2} & \text{LET} \\
\frac{y \in \Delta \quad \Delta, \overline{x}_i \mid \Gamma \vdash e_i \quad \overline{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma \vdash \text{match } y \{ C_i \overline{x}_i \mapsto e_i \}} & \text{BMATCH} \\
\frac{\Delta \mid \Gamma, \overline{x}_i, \varnothing \kappa \vdash e_i \quad k = |\overline{x}_i| \quad \overline{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma \vdash \text{match}! \ e \{ C_i \overline{x}_i \mapsto e_i \}} & \text{DMATCH!}
\end{array}
\]

\[
\begin{array}{ll}
\frac{\Sigma, \overline{y} \vdash e}{\Sigma, f(\overline{y}; \overline{x}) = e} & \text{DEFFUN}
\end{array}
\]

Fig. 4. Well-formed FIP expressions, where the multiplicity of each variable in \( \Gamma \) is 1.

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 \( k \geq 1 \) arguments, is through the \text{REUSE} rule where we need to check well-formedness of each argument, but we also need a reuse credit to guarantee that the needed space is available at evaluation time. The \text{DMATCH!} rule creates such reuse credits: we can destructively match on an owned variable \( x \) to get a reuse credit \( \varnothing \kappa \) 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 match! \( e \{ \ldots \} \) into let \( x = e \) in match! \( x \{ \ldots \} \) 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 \text{REUSE} rule or the \text{EMPTY} rule for zero-sized reuse credits (when an atom is matched).

In contrast, the borrowed match \text{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 \( \Gamma \) cannot be discarded (i.e. freed) or duplicated (i.e. shared), we can temporarily \textit{borrow} them. In the \text{LET} rule the owned environment is split in three parts \( \Gamma_1, \Gamma_2, \Gamma_3 \). The \( \Gamma_1 \) and \( \Gamma_3 \) environments are passed to \( e_1 \) and \( e_2 \) respectively, but the \( \Gamma_2 \) environment is passed to \( e_2 \) as an owned environment, but also to \( e_1 \) as a borrowed environment! Since we \textit{consume} \( \Gamma_2 \) in the derivation of \( e_2 \), we can consider them borrowed in the derivation of \( e_1 \). Note that we still need \( \Gamma_3 \) since \( \Gamma_1 \) is joined with the borrowed \( \Delta \) environment and as such cannot contain any reuse credits (which can thus be included in \( \Gamma_3 \) instead).

The \text{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 \text{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 \( \Gamma_2, \Gamma_3 \) to check \( e_1 \) (where \( \Gamma_3 \) 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 \( x \mapsto C^k \overline{x}_1 \ldots \overline{x}_k \) of size \( k \), or a reuse credit \( \phi_k \) of size \( k \).

Using the eval rule, we can reduce using small steps in the store semantics. The reduction rules have the form \( S \mid e \longrightarrow_s S' \mid e' \) 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 \( \text{reuse}_x \) transition uses a reuse credit \( \phi_k \) in the store to apply the constructor, while \( \text{atom}_x \) allows atoms to be created freely. The \( \text{bmatch}_x \) and \( \text{dmatch}_x \) 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
We assume that we are given stores corresponding to the owned and borrowed values, but only we can now see that any FIP expression can run on the store semantics if we use a store containing where all values have a reference count of one. We can now state the main soundness theorem. We 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 \( \Delta \subseteq \text{dom}(S_1) \), \( S_1 \) sound, \( \Gamma = \text{roots}(S_2) \) and \( S_2 \) linear, then \( [S_1, S_2] e \mapsto^* \bar{\nu} \) implies \( S_1, S_2 \mid e \mapsto^* \{ x \} S_1, S_3 \mid \bar{x} \) where \( [S_3]\bar{x} = \bar{v}, \bar{x} = \text{roots}(S_3) \) and \( S_3 \) is linear.

This is a strong theorem and the proof is quite involved (see Appendix B), both due to destructive update and the ability to match on variables temporarily borrowed in the \( \text{let} \) 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 \( e \mapsto^* \bar{v} \) and \( \emptyset \mid \overline{\alpha} \vdash e \), then \( \overline{\alpha} \mid e \mapsto^* S \mid \bar{x} \) and \( [S]\bar{x} = \bar{v} \).

Furthermore, any FIP function can be called on stores from the environment:

**Corollary 2.**  
If \( \Sigma, f \) is fully-in-place and \( f(\bar{v}_1; \bar{v}_2) \mapsto^* \bar{v} \), and given disjoint linear stores \( S_1, S_2 \) with roots \( \bar{y}, \bar{x} \) and \( [S_1]\bar{y} = \bar{v}_1, [S_2]\bar{x} = \bar{v}_2 \), then also \( S_1, S_2 \mid f(\bar{y}; \bar{x}) \mapsto^* \{ x \} S_1, S_3 \mid \bar{z} \), where \( [S_3]\bar{z} = \bar{v} \).

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 \(|S|\) of a store \( S \) is:  
\[ |\emptyset| = 0 \quad |S, x| = |S| + k \quad |S, x \mapsto C_k x_1 \ldots x_k| = |S| + k. \]

With this definition, we can immediately see that the size of the store doesn’t 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 \( S \mid e \mapsto^* S' \mid e' \), we have \(|S| = |S'|\).

### 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 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 \( \text{drop} \) rule consumes a variable \( x \) from the owned environment. Since the multiplicity of all elements in \( \Gamma \) is still one, this asserts that \( x \) is no longer an element of \( \Gamma \). Similarly, the \( \text{free} \) rule allows discarding a reuse credit.
The operational semantics is also extended with two new reductions for dropping a bound constructor and freeing a reuse credit. With these new rules and reductions for deallocation, the soundness theorem 1 continues to hold (see Appendix B). Again, we can immediately see that the store semantics now only allows deallocation:

**Theorem 3.** (A FBIP program can only deallocate.)

For any $S \mid e \rightarrow^* S' \mid e'$ with the deallocation rules, we have $|S| \geq |S'|$.

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

### 2.5 FIP and FBIP with bounded allocation

We can also model bounded allocation in both the FIP and FBIP calculi to support the `fip(n)` and `fbip(n)` keywords, by adding special allocation credits $\diamond$ to the $\Gamma$ environment. These credits do not have a size, and can be used to allocate a cell of any size. Since we usually have several at a time, we write $\diamond n$ for $n$ such credits.

In the **inst** rule, we can allocate a reuse credit $\diamond_k$ of any size from an allocation credit. We do not have to spend allocation credits, which gives them a weakening rule. We modify the **call** and **deffun** rules from above to respect the `fip(n)` keyword: When calling a function, we require the necessary allocation credits to call $f$. Furthermore, we also require allocation credits for all higher-order functions passed to $f$. Then we can assume in the **bapp** rule that any allocation credits necessary for the function call have already been paid. Finally, in the **deffun** rule, we obtain as many allocation credits as assigned by the keyword.

It is slightly tricky to infer the necessary allocation credits when also inferring deallocation. For example, there can be several, equally valid inferences for expressions that differ only in their use of `free`:

- $x \mid \diamond \vdash \text{match } x \{ \text{True} \mapsto \text{Cons}(\text{Nil}, \text{Nil}); \text{False} \mapsto \text{Nil} \}$
- $x \mid \diamond_2 \vdash \text{match } x \{ \text{True} \mapsto \text{Cons}(\text{Nil}, \text{Nil}); \text{False} \mapsto \text{free } 2; \text{Nil} \}$

The first expression requires an allocation credit, while the second one requires a reuse credit (which is explicitly deallocated). Which one is preferable depends on the availability of a reuse credit $\diamond_2$ in the context of $e$ and on the precise annotation on the function containing $e$: in a `fip(1)` function only the first expression would be valid, while in an `fbip` function only the second one
is allowed (if a reuse credit is available). In our implementation, the Γ context is an output of the inference, but we do not create a joint context for all branches of a match immediately. Instead we keep the Γi contexts of the branches and resolve them as reuse credits become available from surrounding match-statements.

2.6 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 of a mutually recursive group \( \vec{f} \) (which might consist of just \( f \) or more functions). In the call rule we then require two additional conditions. Firstly, \( \Sigma \) contains only functions defined before or as part of the current mutually recursive group. Secondly, we constrain the mutually recursive groups \( \vec{f} \) by requiring that any recursive calls within this group are tail-recursive. Formally, all function definitions \( f \in \vec{f} \) need to be of the form \( f(\vec{y}, \vec{x}) = T[\vec{f}] \):

\[
T[\vec{f}] := e_0 \mid f_i(e_0, e_0) \mid \text{let } \vec{x} = e_0 \in T[\vec{f}] \mid \text{match } e_0 \{ p_i \mapsto T_i[\vec{f}] \} \mid \text{match }! e_0 \{ p_i \mapsto T_i[\vec{f}] \}
\]

wher \( \vec{f} \not\ni \text{fv}(e_0) \). In the call rule, one can pass functions from \( \Sigma \) 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\(^5\), in which the stack usage is always bounded. The \texttt{fip} keyword in our implementation checks if a function is a well-formed FIP\(^5\) function.

In a first-order calculus the previous restrictions immediately imply that the stack size is bounded by the number of mutually recursive groups: any call is either tail-recursive (and executing it does not increase the stack size) or to a previously defined group. However, our FIP calculus is not a first-order calculus since we can pass top-level functions as arguments. Instead, our FIP calculus is a second-order calculus: since we ensure that functions never enter the Γ environment, we cannot return any anonymous function as a result.

In a second-order calculus, it is possible for any top-level function to call back in the list of definitions by calling an anonymous function. However, any sequence of anonymous calls will be to increasingly earlier defined functions. Our stack bound is thus equal to the square of the number of mutually recursive groups.

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 \( \Sigma \) and denote by \(|\epsilon_{\text{max}}|\) the maximum depth of an expression bound in \( \Sigma \). Then we have:

**Theorem 4.** (A FIP program uses constant stack space)

Let \( \Sigma \) be fully-in-place such that for all functions \( f \) in \( \Sigma \) that are mutually recursive with \( \vec{f} \), we have \( f(\vec{y}, \vec{x}) = T[\vec{f}] \). At any intermediate evaluation step \( S \mid f(\vec{y}, \vec{x}) \rightsquigarrow S' \mid E[e'], \) we have \(|E| \leq |\epsilon_{\text{max}}| \cdot |\Sigma|^2\).

This then yields our stack size bound of \(|\Sigma|^2\). The additional factor of \(|\epsilon_{\text{max}}|\) 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 Section Appendix C 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:

```haskell
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:

```haskell
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:

```haskell
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. It is not quite a coincidence that the reuse credits we need in every step line up perfectly – these forms of tail recursive functions can be expressed using a generalisation of zippers known as *dissections* [McBride 2008]. By marking each of these functions as `fip` the compiler checks that they are indeed fully in-place.

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
Translating \( f \) to a tail-recursive \( f' \) function: Applying the zipper:
\[
\begin{align*}
&f(\overline{y}; \overline{x}) = e & f'(\overline{y}; \overline{x}, z) = [e]_z \\
&app(\overline{y}; z, \overline{x}') = \text{match}! z & \text{app}(\overline{y}; z, \overline{x}') = [E_1[\overline{x}']]_{z'}
\end{align*}
\]
\[
\text{zipper} = H | Z_1 z_1 z' | \ldots | Z_n z_n z' \Rightarrow Z_1 z_1 z' \rightarrow [E_1[\overline{x}']]_{z'} \ldots Z_n z_n z' \rightarrow [E_n[\overline{x}']]_{z'}
\]

Tail recursive contexts with \( f \notin \text{fv}(e_0) \):
\[
\begin{align*}
&\text{T} := \Box | \text{let } \overline{x} = e_0 \text{ in } \text{T} | \text{match } e_0 \{ p_i \mapsto T_i \} | \text{match}! e_0 \{ p_i \mapsto T_i \} | \text{drop } \overline{x}, \text{T} | \text{free } k; \text{T}
\end{align*}
\]
Tail recursion translation for a function \( f \) with zipper \( z \):
\[
\begin{align*}
&(\text{tcxt}) [\text{T}[_e_0]_z] = [\text{T}[_e_0]_z] \quad \text{where } f \notin \text{fv}(e_0) \\
&(\text{base}) [\text{e}_0]_z = \text{app}(\overline{y}; z, e_0) \\
&(\text{tail}) [f(\overline{y}; \overline{x}')]_z = f'(\overline{y}; \overline{x}', z) \quad \text{where } \overline{y} \vdash \overline{x}, \overline{z} \vdash E_i[\Box] \text{ and } k = |\overline{z}| + 1
\end{align*}
\]

Fig. 8. TRMReC: tail-recursive modulo reusable contexts.

The standard \( \text{tmap} \) function:
\[
\begin{align*}
\text{fun } \text{tmap}(t : \text{tree}\langle a \rangle, ^a f : a \rightarrow b) : \text{tree}\langle a \rangle \\
\text{match}! t \\
\text{Bin}(1, r) \rightarrow \text{val } l' = \text{tmap}(1,f) \text{ in } l' = \text{tmap}(r,f) \text{ in } \text{Bin}(l', r') \\
\text{Tip}(a) \rightarrow \text{Tip}(f(a))
\end{align*}
\]

Let us focus on the first branch, where a CPS-translation yields the following closures:
\[
\text{Bin}(1, r) \rightarrow \text{tmap}(1, f, \text{fn}(l')\{ \text{tmap}(r,f, \text{fn}(r')(\text{k(Bin}(l', r')))) \}\))
\]

Comparing with our zipper type, we can identify Top with the identity function, Bin\(R \) with the inner closure \( (\text{fn}(r')\ k(\text{Bin}(l', r'))) \), and Bin\(L \) with the outer closure – the zipper is just the defunctionalization of the closures:
\[
\begin{align*}
&\text{fn}(x) x \quad \text{top} \Rightarrow \text{Top} \rightarrow t \\
&\text{fn}(r') k(\text{Bin}(l', r')) \quad \text{Bin}(l', k) \rightarrow \text{app}(\text{Bin}(l', t), f, k) \\
&\text{fn}(l') \text{tmap}(r,f,\text{fn}(r')(\text{k(Bin}(l',r')))\)) \quad \text{Bin}(k,r) \rightarrow \text{down}(r,f,\text{Bin}(t,k))
\end{align*}
\]

The arguments \( r' \) and \( l' \) to the closures correspond to the tree \( t \), the down function to the transformed \( \text{tmap} \) function, and \( \text{app} \) applies the defunctionalized continuation \( k \) to the new tree. As shown by Damvy [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 \( r \) is not included in the zipper, but instead passed directly to \( \text{app} \). This is crucial, since we can not store a borrowed value inside a data structure. Figure 8 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 \( \text{app}(\overline{y}; z, \overline{x}') \) 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 $\left[ e \right]_z$ below. The \textit{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 $T$, 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 \textit{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].

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 $\bar{y}$ (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 $\bar{z}_i$ of $E_i$.

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

\textbf{Theorem 5. (The TRMReC transformation is sound.)}

Let $f$ be a function with $\bar{y} \ x \mapsto f(\bar{y}; \bar{x})$ and $f(\bar{v}_1; \bar{v}_2) \mapsto^* \bar{w}$. If it can be transformed into $f'$, then $\bar{y} \ x \mapsto f'(\bar{y}; \bar{x}, z)$ and $\bar{y} \ x \mapsto \text{app}(\bar{y}; z, \bar{x})$ and $f'(\bar{v}_1; \bar{v}_2, H) \mapsto^* \bar{w}$.

See Appendix D for the proof. We also generalize this theorem to handle recursive calls with varying borrowed arguments and the case where $k \geq |\bar{z}_i| + 1$ (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 $E_1 := \text{let } l' = \square \text{ in } \text{let } r' = \text{tmap}(f \cdot r) \text{ in } \text{Bin } l' \ r'$ we have $f \ | \ \phi_2, r \vdash E_1[\square]$.
- For $E_2 := \text{let } r' = \square \text{ in } \text{Bin } l' \ r'$ we have $f \ | \ \phi_2, l' \vdash E_2[\square]$.

Thus the condition is fulfilled and the translation succeeds.

An important special case of the more general transformation is the tail recursion modulo (product) cons transformation [Bour et al. 2021; Leijen and Lorenzen 2023], where $E = \text{let } \bar{x} = \square \text{ in } \bar{v}$. This can be compiled more efficiently by allocating the values $\bar{v}$ upfront and storing $E$ as pointers to the slots $\bar{x}$ in $\bar{v}$. In our programming examples in section 4 we assume this optimization, but in systems that lack it one can instead obtain a FIP program by using reusable contexts.

\section{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 \$map_\alpha$ to define a (non tail-recursive) \textit{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{align*}
\text{map}(f; x) &= \text{match}! x \\
C^k (x_1 : \tau_1) \ldots (x_k : \tau_k) &\rightarrow \\
&\text{let } y_1 = \$map_{\tau_1}(f; x_1) \text{ in} \\
&\ldots \\
&\text{let } y_k = \$map_{\tau_k}(f; x_k) \text{ in} \\
C^k y_1 \ldots y_k &\rightarrow \\
\text{and} \text{ call the } \$map macro on each of the fields: \\
\$map_T(\alpha)(f; x_1) &= \text{map}(f; x_1) \\
\$map_\alpha(f; x_i) &= f(x_i) \\
\$map_\tau(f; x_i) &= x_i \text{ otherwise} \\
E_i = \text{let } y_i = \square \text{ in } \ldots \text{ in } C^k y_1 \ldots y_k \\
f \ | \ \phi_k, y_1, \ldots, y_{i-1}, x_i+1, \ldots, x_k \vdash E_i[\square]
\end{align*}
FP²: Fully in-Place Functional Programming

The $\text{map}_\tau$ macro dispatches on the type of its argument. If $x$ has type $T \alpha$ it generates a recursive call $\text{map}(f; x)$ if $x$ has type $\alpha$, it generates a call $f(x)$; otherwise it leaves the argument unchanged. In this definition, all recursive calls to $\text{map}$ happen in $\text{map}$, 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 $\text{map}$.

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 $\text{tmap}$). 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 $\text{map}$ or $\text{reverse}$. 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 Appendix A.

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:
This definition is typical for a functional datastructure, but different from the imperative definition as a node does not store a pointer to its parent. Such parent-pointers would not work well in our setting, because they create many small cycles between parent and children, which can never be in-place reused again. Instead, we use pointer-reversal along the insertion path. On this path, the zipper contains the parent links instead of children links and no cycles are created.

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 \texttt{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 \texttt{fixup} function below on them.

But how do we write \texttt{fixup}? Thankfully, we can translate it almost verbatim from the \texttt{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 \texttt{fixup} (by calling \texttt{rebuild} defined below) as the parent node was colored black.

\begin{verbatim}
  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(Node(Red,left-rotate(Node(Red,zpl,zpk,zpv,z.set-black)),zppk,zppv,y))) // (2)
        _ -> rebuild(zp, z)
  // and cases as above with "left" and "right" interchanged

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

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

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

The last remaining function is \texttt{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.
FP²: Fully in-Place Functional Programming

```haskell
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:

\[
\begin{align*}
[4,3,2,1] & \rightarrow [[4],[3],[2],[1]] \quad \text{// create sorted singleton lists} \\
& \rightarrow [[3,4],[1,2]] \rightarrow [1,2,3,4] \quad \text{// merge pairs of sorted lists} \\
& \rightarrow [1,2,3,4] \quad \text{// extract sorted list}
\end{align*}
\]

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 `Ones` or `Subs` 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`.

```haskell
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)
End
```

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:

\[
\begin{align*}
\text{Cons}(4,\text{Cons}(3,\text{Cons}(1,\text{Nil}))) & \quad \text{// start with unsorted list} \\
& \rightarrow \text{One}(4,\text{One}(3,\text{One}(1,\text{End}))) \quad \text{// created sorted singleton lists} \\
& \rightarrow \text{Sub}(\text{Nil2}(3,4),\text{Sub}(\text{Nil2}(1,2),\text{End})) \quad \text{// merge pairs of sorted lists} \\
& \rightarrow \text{Sub}(\text{Cons2}(1,\text{Cons2}(2,\text{Nil2}(3,4))),\text{End}) \quad \text{// ... until only one list2 is left} \\
& \rightarrow \text{Cons}(1,\text{Cons}(2,\text{Cons}(3,\text{Cons}(4,\text{Nil})))) \quad \text{// convert back to list}
\end{align*}
\]

Note that this does not solve the (much harder) in-place sorting problem for arrays, where we can not distinguish sublists through tags and shuffling pointers.

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]:

```haskell
fip fun quicksort(xs : list<a>)
    match! xs
    Nil -> Nil
    Cons(pivot, xx) ->
    val (lo, hi) = split(pivot, xx) \quad \text{// split xx into (lo,hi) in-place} \\
    val (lo',hi') = (quicksort(lo),quicksort(hi)) \quad \text{// 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:
The problem here is that the zipper needs to store \(\text{pivot}, \text{hi}\) 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 \(\text{hi}\) 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 \(\text{hi}\) either not at all (if \(\text{hi}\) is empty), as a singleton (if \(\text{hi}\) is a one-element list) or else as a list2. We can pass the parent zipper into the \text{split} function, which now returns a list \(\text{lo}\) and a partition \(\text{hi}\) which includes the zipper. Then we obtain a fully in-place solution:

\[
\begin{align*}
\text{Cons}(\text{pivot}, \text{xx}) & \rightarrow & \text{// reuse credit of size 2 available} \\
\text{val} & \text{(lo}, \text{hi}) = \text{split}(\text{pivot}, \text{xx}) \\
\text{quicksort'}(\text{lo}, \text{Z1}(\text{pivot}, \text{hi}, \text{zipper})) & \text{ // reuse credit of size 3 needed}
\end{align*}
\]

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 \text{cons} function on finger trees does not appear to be FIP: only the \text{More} constructors can be reused as the other datatypes do not match up for reuse. We can fix this, however, by \text{padding} all constructors with a dummy atom \text{Pad} so that they all have three slots.

We use Claessen [2020]'s version of finger trees, since Hinze and Paterson's version has an extra \text{Four} constructor which would require us to pad all \text{cons} to four fields.

\[
\begin{align*}
\text{fun} & \text{\ cons(x : a, s : seq<a>)) : seq<a>} \\
\text{match} & \text{! s} \\
\text{Empty} & \rightarrow \text{Unit(x, Pad, Pad)} \\
\text{Unit(y, _, _)} & \rightarrow \text{More(One(x, Pad, Pad), Empty, One(y, Pad, Pad))} \\
\text{More(One(y, _, _), q, u)} & \rightarrow \text{More(Two(x, y, Pad), q, u)} \\
\text{More(Two(y, z, _), q, u)} & \rightarrow \text{More(Three(x, y, z), q, u)} \\
\text{More(Three(y, z, w), q, u)} & \rightarrow \text{More(Two(x, y, Pad), cons(Pair(z, w, Pad), q), u)}
\end{align*}
\]

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

Therefore, we pair a finger tree \text{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.

\[
\text{type buffer} \ {\{ BEmpty; BCons(next : buffer, b : pad, c : pad) \}}
\]

We then pass the necessary reuse credits of size 3 to \text{cons}, which we either use to create a new cell in the finger tree or fill up the buffer. If we recurse into \text{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 \text{More(One(x, Pad, Pad), Empty, One(y, Pad, Pad))}, so it would seem that we need to pass at least two credits. But that would mean that we need \(6n\) space to represent \(n\) elements in a finger tree! We can do better by specializing \text{More(One)} as \text{More0} to represent the two-element list as \text{More0}(x, \text{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 \text{cons} function than takes a reuse credit \text{unit3} and becomes:
Extended syntax:

\[
\begin{align*}
\text{v} & ::= \ldots | \lambda x. e \\
\text{e} & ::= \ldots | \text{dup } x; e | \text{dropru } x; e | \text{alloc } k; e
\end{align*}
\]

Extended evaluation steps:

\[
\begin{align*}
\text{(beta)} & \quad (\lambda x. e) \bar{x} \rightarrow e[\bar{x}:=\bar{v}] \\
\text{(dup)} & \quad \text{dup } x; e \rightarrow e \\
\text{(dropru)} & \quad \text{dropru } x; e \rightarrow e \\
\text{(alloc)} & \quad \text{alloc } k; e \rightarrow e
\end{align*}
\]

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 9 formalizes the \(\lambda^\text{fp}\) calculus as an extension of the syntax and operational semantics of the FBIP calculus (in Figure 4 and 6). It has full lambda expressions \(\lambda x. e\) now since arbitrary allocation is allowed. Here we write the free variables of the lambda expression explicitly as (the multiset) \(\bar{z}\). 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 \(\text{dup } x\); \(e\) and \(\text{dropru } x; e\) expressions that let us duplicate owned variables and explicitly reuse dropped variables. Finally, the \(\text{alloc } k; e\) allows for arbitrary allocation of constructors by creating reuse credits \(\circ_k\) at runtime.

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

\[\text{FP}^2: \text{Fully in-Place Functional Programming}\]
in the heap semantics).

We rephrase the original Perceus linear resource calculus, called $\lambda^1$ [Reinking, Xie et al. 2021], in Figure 9 as a type system (instead of a typed translation). We call any expressions in $\Delta \mid \Gamma \vdash e$ 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 $\Gamma$ environment, but unlike a pure linear logic it now has an escape hatch: through rules like $\text{dup}$ we can freely duplicate "linear" variables by maintaining reference counts dynamically at runtime. As we can see, there is a surprisingly close connection between $\lambda^{fip}$ and the FBIP and FIP calculi where each one is a strict subset of the other: $\text{FIP} \subset \text{FBIP} \subset \lambda^{fip}$. As such, the FIP calculus is exactly the subset of $\lambda^{fip}$ that excludes the rules that require dynamic reference counting!

The $\text{dup}$ rule is the rule that either allows us to use a borrowed variable ($x \in \Gamma$) as owned, or duplicates an owned variable ($x \in \Delta$). The $\text{alloc}$ rule now allows arbitrary allocation of a constructor by adding a reuse credit $\circ k$ to the owned environment. With the full lambda expressions, we also have an $\text{app}$ 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 $\Gamma_2$ in the $e_1$ derivation just like our earlier $\text{let}$ rule in Figure 4. We refrain from doing that here for simplicity as we can always use $\text{let}$ if borrowing is required.

The $\text{lam}$ 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 $\text{call}$ rule).

The $\text{match}$ rule can match on any borrowed or owned variable. However, each branch must start by dupping the matched constructor fields (as $\text{dup}(\bar{x}_i)$). 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 $\text{match}$ rule can only match variables but we can always rewrite an expression match $e \{ \ldots \}$ into let $x \equiv e$ in match $x \{ \ldots \}$ for a fresh $x$ when required. Since $\text{match}$ no longer creates reuse credits, we can now create them explicitly instead using the "drop reuse" $\text{dropru}$ rule. This drops a variable $x$, and immediately allows for a reuse credit $\circ k$ where $k$ is the allocated size of $x$.

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

\[
\text{match! } x \{ C_i \bar{x}_i \to e_i \} \quad \text{with} \quad \text{match } x \{ C_i \bar{x}_i \to \text{dup } \bar{x}_i ; \text{dropru } x ; e_i \} .
\]

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

\[
\frac{}{\Delta \mid \Gamma, \bar{x}_i, \circ k \vdash e_i \ (1) \quad \text{DMATCH!}}
\]

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

\[
\frac{}{\Delta \mid \Gamma, x \vdash \text{match } x \{ C_i \bar{x}_i \to \text{dup } \bar{x}_i ; \text{dropru } x ; e_i \} \quad \text{MATCH}}
\]

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 $\lambda^{fip}$ 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...
FP²: Fully in-Place Functional Programming

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

\[
H ::= \emptyset \mid H, o_k \mid H, x \mapsto^n \varphi \quad \text{where } \varphi ::= C \overline{x} \mid \lambda \overline{x}. e
\]

The \( \rightarrow_h \) relation extends the \( \rightarrow_s \) (using \( H \) for \( S \)) relation:

\[
\begin{align*}
(dup_h) & \quad H, x \mapsto^n v \quad \text{dup } x; e \rightarrow_h H, x \mapsto^{n+1} v \quad | \quad e \\
(dlam_h) & \quad H, x \mapsto^1 \lambda \overline{x}. e' \quad \text{drop } x; e \rightarrow_h H \quad | \quad \text{drop } \overline{x}; e \\
(drop_h) & \quad H, x \mapsto^{n+1} v \quad \text{drop } x; e \rightarrow_h H, x \mapsto^n v \quad | \quad e \quad \text{(if } n \geq 1) \\
(dropru_h) & \quad H, x \mapsto^n C \overline{x} \quad \text{dropru } x; e \rightarrow_h H, o_k, x \mapsto^n C \overline{x} \quad | \quad e \quad \text{(if } n \geq 1) \\
(daconru_h) & \quad H, x \mapsto^1 C \overline{x} \quad \text{dropru } x; e \rightarrow_h H, o_k \quad | \quad \text{drop } \overline{x}; e \\
(aloc_h) & \quad H \mid \text{alloc } k; e \rightarrow_h H, o_k \quad | \quad e \\
(lam_h) & \quad H \mid \lambda \overline{x}. e \rightarrow_h H, x \mapsto^1 \lambda \overline{x}. e \mid x \quad \text{(fresh } x) \\
(app_h) & \quad H \mid (y) \overline{x} \rightarrow_h H \mid \text{dup } \overline{x}; \text{drop } y; e[\overline{x}:=\overline{y}] \quad (y \mapsto^n \lambda \overline{x}. e \in H)
\end{align*}
\]

Fig. 10. Heap semantics of \( \lambda^{fip} \) – extending the FBIP store semantics as shown in Figure 5 and 6.

5.1 Elaborating borrowing

However, the elaboration needs to take some care when functions with borrowed parameters are passed as arguments. For example, what should happen when the user passes \( \text{is-red} \) to \( \text{map} \):

\[
\begin{align*}
\text{map}(\text{is-red}, [t1, t2, t3]) : \text{list<bool>} \\
\text{fip fun is-red(\text{\textasciitilde t} : \text{tree}) : bool} \\
\text{fip fun map( l : list<\text{\textasciitilde a, } \text{\textasciitilde f} : \text{a -> b } ) : list<\text{\textasciitilde b}>}
\end{align*}
\]

Here, \( \text{map} \) assumes that the given function \( f \) takes its argument as owned, while \( \text{is-red} \) takes its argument \( t \) as borrowed. In a linear type system (and the FIP calculus), such a program would be rejected. However, we follow a different approach here: Since we seek to integrate fully in-place programs seamlessly into a general-purpose functional programming language, the program above should be accepted. Instead, we \( \text{wrap} \) functions with borrowed parameters, into a closure that takes only owned parameters, calls the original function and then drops as necessary [Ullrich and de Moura 2019]. Formally, we can derive:

\[
\begin{array}{c}
\frac{\overline{y} \mid \overline{x} \vdash \overline{x}}{\emptyset \mid z \vdash z} \quad \text{CALL} \\
\frac{\emptyset \mid \overline{y}, \overline{x} \vdash f(\overline{y};\overline{x})}{\emptyset \mid \overline{y} : \vdash \overline{z} \quad \text{drop } \overline{y}; z} \quad \text{DROP} \\
\frac{\emptyset \mid \overline{y}, \overline{x} \vdash \text{let } z = f(\overline{y};\overline{x}) \text{ in drop } \overline{y}; z}{\emptyset \mid \overline{y}, \overline{x} \vdash \text{let } z = f(\overline{y};\overline{x}) \text{ in drop } \overline{y}; z} \quad \text{LET} \\
\frac{\Delta \mid \emptyset \vdash \lambda \overline{x}. (\overline{y}, \overline{x}). \text{let } z = f(\overline{y};\overline{x}) \text{ in drop } \overline{y}; z}{\Delta \mid \emptyset \vdash \lambda \overline{x}. (\overline{y}, \overline{x}) \quad \text{LET}}
\end{array}
\]

This wrapping can be done cheaply for top-level functions with borrowed parameters, since the new closure has no free variables and can be lambda-lifted to the top-level without allocation. Later compilation stages may then decide to inline the nested function call – making the wrapping zero-overhead at the cost of code duplication.

However, this would be more expensive on closures, where \( f \) itself is a free variable! Then we would have to allocate the wrapping closure and could not rely on inlining to remove the overhead of the extra indirection. For this reason, borrowing is usually not done on closures – and we reflect this in our FIP and \( \lambda^{fip} \) calculi to remain compatible with this interpretation of reference counting.

5.2 Heap semantics

Figure 10 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 \( n \geq 1 \) for each binding. The heap now contains reuse credits \( o_k \), a constructor binding \( x \mapsto^n C \overline{x} \), or a closure \( x \mapsto^n \lambda \overline{x}. e \).
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:

\[(bmatch_h) \quad H, \ y \mapsto^n C_k \vec{y} \mid \text{match } y \{ p \mapsto e \} \quad \rightarrow_h \quad H, \ y \mapsto^n C_k \vec{y} \mid e_i[\vec{x}:=\vec{y}] \quad (p_i = C_k \vec{y})\]

\[(dmatch_h) \quad H, \ x \mapsto^1 C_k \vec{y} \mid \text{match! } x \{ p \mapsto e \} \quad \rightarrow_h \quad H, \ x \mapsto^1 C_k \vec{y} \mid \phi_k \mid e_i[\vec{x}:=\vec{y}] \quad (p_i = C_k \vec{x})\]

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 \( \vec{z} \), 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 \( \vec{z} \) 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 \((deconru_h)\) rule and the \((decon)\) rule in Figure 6. The \((deconru)\) rule creates a reuse credit if the reference count is unique, while the \((dropru)\) 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:

\[
\begin{align*}
\text{H}, \ x \mapsto^1 C_k \vec{y} \mid \text{match! } x \{ C_i \vec{x}_i \mapsto e_i \} & \quad \rightarrow_h \quad \text{H}, \ x \mapsto^1 C_k \vec{y} \mid \text{match } x \{ C_i \vec{x}_i \mapsto \text{dup } \vec{x}_i; \ dropru x; \ e_i \} \\
\text{H}, \ x \mapsto^1 C_k \vec{y} \mid \text{match } x \{ C_i \vec{x}_i \mapsto \text{dup } \vec{x}_i; \ dropru x; \ e_i \} & \quad \rightarrow_h \quad \text{H}, \ x \mapsto^1 C_k \vec{y} \mid \text{dup } \vec{y}; \ dropru x; \ e_i[\vec{x}_i:=\vec{y}] \\
\text{H}, \ x \mapsto^1 C_k \vec{y} \mid \text{dup } \vec{y}; \ dropru x; \ e_i[\vec{x}_i:=\vec{y}] & \quad \rightarrow_h \quad \text{H}, \ x \mapsto^1 C_k \vec{y} \mid \text{drop } \vec{y}; \ e_i[\vec{x}_i:=\vec{y}] \\
\text{H}, \ x \mapsto^1 C_k \vec{y} \mid \text{drop } \vec{y}; \ e_i[\vec{x}_i:=\vec{y}] & \quad \rightarrow_h \quad \text{H}, \ x \mapsto^1 C_k \vec{y} \mid e_i[\vec{x}_i:=\vec{y}]
\end{align*}
\]

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:

\[
\begin{align*}
\ldots & \quad \rightarrow_h \quad \text{H}', \ x \mapsto^{n+1} C_k \vec{y} \mid \text{drop } x; \ e_i[\vec{x}_i:=\vec{y}] \quad \{ \text{H’ is H but with all } \vec{y} \text{ refcounts } +1 \} \quad (A) \\
\text{H}', \ x \mapsto^{n+1} C_k \vec{y} \mid \text{drop } x; \ e_i[\vec{x}_i:=\vec{y}] & \quad \rightarrow_h \quad \text{H}', \ x \mapsto^{n+1} C_k \vec{y} \mid \text{drop } \vec{y}; \ e_i[\vec{x}_i:=\vec{y}] \\
\text{H}', \ x \mapsto^{n+1} C_k \vec{y} \mid \text{drop } \vec{y}; \ e_i[\vec{x}_i:=\vec{y}] & \quad \rightarrow_h \quad \text{H}', \ x \mapsto^{n+1} C_k \vec{y} \mid e_i[\vec{x}_i:=\vec{y}]
\end{align*}
\]

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.

### 5.3 Soundness of the Heap Semantics

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

**Definition 3.** \((\text{Heap Soundness and Linearity})\)

For a heap \( H \) we write \( \text{dom}(H) \) to denote the set of variables \( x \) bound in \( H \) and write \( \text{rng}(H) \) to denote the set of values \( C \vec{x} \) 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 \textit{sound} if all free variables in
FP²: Fully in-Place Functional Programming

\( \text{rng}(H) \) are bound: \( \text{fv}(\text{rng}(H)) \subseteq \text{dom}(H) \). A heap is linear if it is sound, and any variable \( x \mapsto^n v \) in \( \text{dom}(H) \) occurs at most \( n \) times in the free variables of \( \text{rng}(H) \). By \( \text{roots}(H) \) we denote the multi-set of reuse credits of \( H \) and variables \( x \mapsto^n v \) of \( \text{dom}(H) \), which contains any variable \( n - m \) times, if it occurs \( m \) times in the free variables of \( \text{rng}(H) \).

The definition of linearity ensures that mutation is safe if the reference count is one. Exactly as in the store semantics, we write \([H]\overline{x}\) 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:

\[
\begin{align*}
&\emptyset \otimes H_2 = H_2 \\
&H_1, \circ_k \otimes H_2 = H_1 \otimes H_2, \circ_k \\
&H_1, x \mapsto^n v \otimes H_2 = H_1 \otimes H_2, x \mapsto^n v \quad \text{iff } x \notin \text{dom}(H_2) \\
&H_1, x \mapsto^n v \otimes H_2, x \mapsto^m v, z \mapsto^{k+1} w = H_1 \otimes H_2, x \mapsto^{n+m} v, z \mapsto^k w \quad \text{iff } \overline{z} = \text{fv}(v)
\end{align*}
\]

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 [Jung et al. 2015] where every result is valid. However, we prefer a categorical view on linear heaps where a morphism \( H_1 \to H_2 \) exists if \( \text{dom}(H_1) \subseteq \text{dom}(H_2) \) and \( \text{roots}(H_1) \subseteq \text{roots}(H_2) \). 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 then show that heap evaluation leaves the borrowed values unchanged:

**Theorem 6.** *(The heap semantics is sound for well-formed Perceus programs)*

If \( \Delta \vdash \Gamma \vdash e \) and given heaps \( H_1, H_2 \) with \( \Delta \subseteq \text{dom}(H_1) \), \( H_1 \) sound, \( \Gamma = \text{roots}(H_2) \) and \( H_2 \) linear, then \( [H_1, H_2]e \mapsto^* \overline{v} \) implies \( H_1 \otimes H_2 \vdash e \mapsto^*_H H_1 \otimes H_2 \mid \overline{v} \) where \( [H_3] \overline{x} = \overline{v}, \overline{x} = \text{roots}(H_3) \) 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 \( (\overline{x} = \text{roots}(H_3)) \). Our proof (in Appendix E) is novel and may be well suited to possible mechanized formalization. As a corollary, any closed \( \lambda^{\text{fip}} \) expression can evaluate starting from an empty heap:

**Corollary 3.**

If \( e \mapsto^* \overline{v} \) and \( \emptyset \vdash e \), then \( \emptyset \vdash e \mapsto^*_H H \mid \overline{x} \) and \([H] \overline{x} = \overline{v} \).

While it is outside the scope of this paper, we could also modify the let rule of our calculus with a \((*)\)-condition to characterize garbage-free and frame-limited derivations [Lorenzen and Leijen 2022]. However, borrowing makes it harder to achieve these properties 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 11 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 v2.4.1-dev-fip. 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.
Anton Lorenzen, Daan Leijen, and Wouter Swierstra

Fig. 11. Benchmarks on Ubuntu 22.04.2 (AMD 7950x), Koka v2.4.1-dev-fbip.

- **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 the one in Section 4.1, 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** variant uses 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 such that 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 \textit{fip} variant will still reuse the zipper it uses to traverse the tree. This is also shows why \textit{std} and \textit{std-reuse} are performing the similarly since there is no reuse possible for the standard algorithms. That \textit{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 environment, 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 \texttt{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 \(n \times n\) space in Hofmann’s calculus.

While we only model unique and borrowed values in our FIP calculus, \textit{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 $\Gamma_2$ in \texttt{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.

Wadler [Wadler 1984] describes the \textit{listless} transformation that can transform compositions of many common list functions, like \texttt{map}, \texttt{append}, or \texttt{partition}, into a state machine that does not allocate any \texttt{Cons} cells. Functions that cannot be transformed include our \texttt{reverse} example, sorting a list, or handle trees. Since the transform can discard \texttt{Cons} cells, we believe that any listless program is also a well-formed FBIP program.

8 \hspace{1em} 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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. ACM SIGPLAN Notices 50 (1). ACM New York,
Anton Lorenzen, Daan Leijen, and Wouter Swierstra

NY, USA: 637–650.


FP²: Fully in-Place Functional Programming

A  EXAMPLE CODE
This section contains the full examples discussed in Section 4. These examples do not use match!, drop or free as they are automatically inferred by our implementation. We also write val (C x1 ... xn) = x in e for match x ( C x1 ... xn -> e ) if the match has just a single case.

A.1  Red-Black Trees

```plaintext
import std/num/int32

type color
  Red
  Black

type tree
  Node(color : color, lchild : tree, key : int32, value : bool, rchild : tree)
  Leaf

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

type accum
  Done
  NodeL(color : color, lchild : accum, key : int32, value : bool, rchild : tree)
  NodeR(color : color, lchild : tree, key : int32, value : bool, rchild : accum)

fip() fun ins(t : tree, key : int32, v : bool, z : accum) : exn tree
  match t
    Node(c, l, kx, vx, r)
      -> if key < kx then ins(l, key, v, NodeL(c, z, kx, vx, r))
      elif key > kx then ins(r, key, v, NodeR(c, l, kx, vx, z))
      else balance(z, Node(c, l, key, v, r))
    Leaf -> balance(z, Node(Red, Leaf, key, v, Leaf))

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

fip fun rebuild(z : accum, t : tree) // Turn the zipper into a tree without rotating
  match z
    NodeR(c, l, k, v, z1) -> rebuild(z1, NodeR(c, l, k, v, t))
    NodeL(c, z1, k, v, r) -> rebuild(z1, NodeL(c, t, k, v, r))
    Done -> t

fip() fun insert(t : tree, k : int32, v : bool) : <exn> tree
  ins(t, k, v, Done)

fip fun balance(z : accum, t : tree ) : exn tree
  match z
    NodeRed(l1, k1, v1, z1) -> match z1
      NodeRed(_, l2, k2, v2, z2) -> // black
        if is-red(l2) then balance(z2, Node(Red(l2.set-black, k2, v2, Node(Black, l1, k1, v1, t)))
        else rebuild(z2, Node(Black, Node(Red(l2, k2, v2, l1), k1, v1, t))
    NodeRed(_, z2, k2, v2, r2) -> // black
      if is-red(r2) then balance(z2, Node(Red, Node(Black, l1, k1,v1,t), k2, v2, r2.set-black))
      else match t
        Node(_, l, k, v, r) -> rebuild(z2, Node(Black, Node(Red(l1,k1,v1,l1), k, v, Node(Red(r,k2,v2,r2))))
      Done -> Node(Black, l1, k1, v1, t)
    NodeRed(z1, k1, v1, r1) -> match z1
    NodeRed(_, z2, k2, v2, r2) -> // black
      if is-red(r2) then balance(z2, Node(Red, Node(Black, t, k1, v1, r1), k2, v2, r2.set-black))
      else rebuild(z2, Node(Black, t, k1, v1, Node(Red(r1,k2,v2,r2))))
```

32
A.2 Stable Merge Sort

Derived from the merge sort of Haskell’s Data.List.sort function.

```plaintext
import std/num/int32
import std/os/env

alias elem = int32

ref type pad
  Pad

type unit2
  Unit2(a : pad, b : pad)

type pair<a>
  Pair(a : a, b : a)

type sublist<a>
  SCons(a : a, cs : sublist<a>)
  STuple(a : a, b : a)

type partition<a>
  Sublist(c : sublist<a>, z : partition<a>)
  Singleton(c : a, z : partition<a>)
  End

def reverse-go(c : sublist<a>, acc : sublist<a>, u : unit2) : sublist<a>
  match c
  SCons(a, cs) -> reverse-go(cs, SCons(a, acc), u)
  STuple(a, b) -> SCons(b, SCons(a, acc))

def reverse-sublist(c : sublist<a>) : sublist<a>
  match c
  SCons(a, SCons(b, c)) -> reverse-go(c, STuple(b, a), Unit2(Pad, Pad))
  SCons(a, STuple(b, c)) -> SCons(c, STuple(b, a))
  STuple(a, b) -> STuple(b, a)

def sequences(xs : list<elem>) : div partition<elem>
  match(xs)
  Cons(a, Cons(b, xs1)) -> if(a > b)
    then
      val (sublist, bs) = descending(b, STuple(b, a), xs1)
      Sublist(sublist, sequences(bs))
    else
      val (sublist, bs) = ascending(b, STuple(b, a), xs1)
      Sublist(sublist, sequences(bs))
  Cons(a, Nil) -> Singleton(a, End)
  Nil -> End

def descending(a : elem, sublist : sublist<elem>, bs : list<elem>) : (sublist<elem>, list<elem>)
  match(bs)
  Cons(b, bs1) | a > b -> descending(b, SCons(b, sublist), bs1)
  bs -> (sublist, bs)
```

FP²: Fully in-Place Functional Programming

fip fun ascending(a : elem, sublist : sublist<elem>, bs : list<elem>) : (sublist<elem>, list<elem>)
match bs
  Cons(b, bs1) | (a <= b) -> ascending(b, SCons(b, sublist), bs1)
  bs -> (reverse-sublist(sublist), bs)

fip fun to-list(c : sublist<elem>, u : unit2) : list<elem>
match c
  SCons(a, cs) -> Cons(a, to-list(cs, u))
  STuple(a, b) -> Cons(a, Cons(b, Nil))

fip fun merge-all(xs : partition<elem>) : list<elem>
match(xs)
  Sublist(x, End) -> to-list(x, Unit2(Pad,Pad))
  Singleton(x, End) -> Cons(x, Nil)
  xs -> merge-all(merge-pairs(xs))

fip fun merge-pairs(xs : partition<elem>) : partition<elem>
match(xs)
  Sublist(a, Sublist(b, xs1)) -> Sublist(merge(a, b, Unit2(Pad,Pad)), merge-pairs(xs1))
  Sublist(a, Singleton(b, xs1)) -> Sublist(merge-last-left(a, b, Unit2(Pad,Pad)), merge-pairs(xs1))
  Singleton(a, Sublist(b, xs1)) -> Sublist(merge-last-right(a, b, Unit2(Pad,Pad)), merge-pairs(xs1))
  Singleton(a, Singleton(b, xs1)) ->
    Sublist(if a <= b then STuple(a, b) else STuple(b, a), merge-pairs(xs1))
  xs -> xs

fip fun merge(c1 : sublist<elem>, c2 : sublist<elem>, u : unit2) : sublist<elem>
match c1
  SCons(a, cs1) -> match c2
    SCons(b, cs2) ->
      if a <= b then SCons(a, merge(cs1, SCons(b, cs2), u))
      else SCons(b, merge(SCons(a, cs1), cs2, u))
    STuple(b, c) ->
      if a <= b then SCons(a, merge(cs1, STuple(b, c), u))
      else SCons(b, merge-last-left(SCons(a, cs1), c, u))
  STuple(a, b) -> match c2
    SCons(c, cs2) ->
      if a <= c then SCons(a, merge-last-right(b, SCons(c, cs2), u))
      else SCons(c, merge(STuple(a, b), cs2, u))
    STuple(c, d) ->
      if a <= c then SCons(a, merge-right(b, Pair(c, d), u))
      else SCons(c, merge-left(Pair(a, b), d, u))

fip fun merge-last-right(a : elem, c2 : sublist<elem>, u : unit2) : sublist<elem>
match c2
  SCons(b, cs2) -> if a <= b then SCons(a, SCons(b, cs2))
  else SCons(b, merge-last-right(a, cs2, u))
    STuple(b, c) -> merge-right(a, Pair(b, c), u)

fip fun merge-last-left(c2 : sublist<elem>, d : elem, u : unit2) : sublist<elem>
match c2
  SCons(a, cs2) -> if a <= d then SCons(a, merge-last-left(cs2, d, u))
  else SCons(d, SCons(a, cs2))
    STuple(a, b) -> merge-left(Pair(a, b), d, u)

fip fun merge-right(a : elem, p : pair<elem>, u : unit2) : sublist<elem>
match p
  Pair(b, c) -> if a <= b then SCons(a, STuple(b, c))
  else SCons(b, if a <= c then STuple(a, c) else STuple(c, a))

fip fun merge-left(p : pair<elem>, d : elem, u : unit2) : sublist<elem>
match p
  Pair(a, b) -> if a <= d then SCons(a, if b <= d then STuple(b, d) else STuple(d, b))
  else SCons(d, STuple(a, b))
### A.3 Quick Sort

```asciidoc
import std/num/int32
import std/os/env

alias elem = int32

ref type pad
    Pad

ref type unit2
    Unit2(a : pad, b : pad)

type maybe2<a>
    Nothing2
    Just2(a : a, b : pad)

type sublist<a>
    SCons(a : a, cs : sublist<a>)
    STuple(a : a, b : a)

type partition<a>
    Sublist(c : sublist<a>, bdl : partition<a>)
    Singleton(c : a, bdl : partition<a>)
    End

fip fun from-list(x, y, u : unit2, xs : list<elem>) : sublist<elem>
    match xs
    Cons(z, zs) -> SCons(x, from-list(y, z, u, zs))
    Nil -> STuple(x, y)

fip fun to-bundle(xs : list<elem>, bdl' : partition<elem>) : partition<elem>
    match xs
    Cons(x, Cons(y, xs)) -> Sublist(from-list(x, y, Unit2(Pad, Pad), xs), bdl')
    Cons(x, Nil) -> Singleton(x, bdl')
    Nil -> bdl'

fip fun quicksort(xs : list<elem>) : div list<elem>
    quicksort-go(to-bundle(xs, End))

fip fun quicksort-go(bdl : partition<elem>) : div list<elem>
    match bdl
    Sublist(xs, bdl') -> match xs
    SCons(p, xs) ->
        val (lo, hi) = partition(p, xs, Unit2(Pad, Pad))
        quicksort-go(to-bundle(lo, Singleton(p, to-bundle(hi, bdl'))))
        STuple(x, y) | x <= y -> Cons(x, Cons(y, quicksort-go(bdl')))
        | _ -> Cons(y, Cons(x, quicksort-go(bdl')))
    Singleton(p, b) -> Cons(p, quicksort-go(b))
    End -> Nil

fip fun partition(^p : elem, xs : sublist<elem>, u : unit2) : (list<elem>, list<elem>)
    match(xs)
    SCons(x, xx) -> if p <= x
        then val (lo, hi) = partition(p, xx, u)
        (lo, Cons(x, hi))
        else val (lo, hi) = partition(p, xx, u)
        (Cons(x, lo), hi)
    STuple(x, y) -> if p <= x
        then if p <= y
            then (Nil, Cons(x, Cons(y, Nil)))
            else (Cons(y, Nil), Cons(x, Nil))
        else if p <= y
            then (Cons(x, Nil), Cons(y, Nil))
            else (Cons(x, Cons(y, Nil)), Nil)
```
A.4  Finger Trees

// Adapted from "Finger Trees Explained Anew, and Slightly Simplified (Functional Pearl)", Claessen
import std/num/int32
ref type pad
Pad
type reuse3
  Reuse3(a : pad, b : pad, c : pad)
type afew<a>
  One(a : a, b : pad, c : pad)
  Two(a : a, b : a, c : pad)
  Three(a : a, b : a, c : a)
type tuple<a>
  Pair(a : a, b : a, c : pad)
  Triple(a : a, b : a, c : a)
type seq<a>
  Empty
  Unit(a : a, b : pad, c : pad)
  More0(l : a, s : seq<tuple<a>>, r : afew<a>)
  More(l : tuple<a>, s : seq<tuple<a>>, r : afew<a>)
type buffer
BNil
BCons(next : buffer, b : pad, c : pad)
value type bseq<a>
  BSeq(s : seq<a>, q : buffer)
// Isomorphic to (,,,) but unboxed
value type tuple4<a,b,c,d>
  Tuple4(fst:a,snd:b,thd:c,field4:d)
fun bhead("bs : bseq<a>") : exn a
match bs
  BSeq(s, _) -> head(s)
fun head("s : seq<a>") : exn a
match s
  Unit(x) -> x
  More0(x, _, _) -> x
  More(Pair(x, _, _), _, _) -> x
  More(Triple(x, _, _), _, _) -> x
fip fun bcons(x : a, u3 : reuse3, bs : bseq<a>) : exn bseq<a>
val BSeq(s, b) = bs
val (s', b') = cons(x, u3, s, b)
BSeq(s', b')

fip fun cons(x : a, u3 : reuse3, s : seq<a>, b : buffer) : exn (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)
  More(Pair(y, z, _), q, u) ->
    (More(Triple(x, y, z), q, u), BCons(b, Pad, Pad))
  More(Triple(y, z, w), q, u) ->
    val BCons(b', _, _) = b
    val (q', b'') = cons(Pair(z, w, Pad), u3, q, b')
    (More(Pair(x, y, Pad), q', u), b'')

fip fun buncons(bs : bseq<a>) : exn (a, reuse3, bseq<a>)
val BSeq(s, b) = bs
val Tuple4(x, u3, s', b') = uncons(s, b)
(x, u3, BSeq(s', b'))
fip fun uncons\(s : \) seq\(\langle a \rangle\), \(b : \) buffer\) : exn tuple4\(\langle a \rangle\), reuse3, seq\(\langle a \rangle\), buffer\)
match s
\(\) Unit\(x, _ , _ \) ->
\(\) Tuple4\(x, \) Reuse3\(Pad,Pad,Pad\), Empty, b
\(\) More\(Triple\(x, y, z\), q, u \) ->
\(\) val BCons\(b', _ , _ \) = b
\(\) Tuple4\(x, \) Reuse3\(Pad,Pad,Pad\), More\(Pair\(y, z, Pad\), q, u \), b'
\(\) More\(Pair\(x, y, _\), q, u \) ->
\(\) Tuple4\(x, \) Reuse3\(Pad,Pad,Pad\), More\(Pair\(y, z, Pad\), q, u \), b
\(\) More\(0\(x, q, u \) \) ->
\(\) val \(q', b'\) = more\(q, u \), b
\(\) Tuple4\(x, \) Reuse3\(Pad,Pad,Pad\), q', b'

fip fun more\(q : \) seq\(\langle tuple\langle a \rangle \rangle\), \(u : \) afew\(\langle a \rangle\), \(b : \) buffer\) : exn (seq\(\langle a \rangle\), buffer)
match q
\(\) Empty \rightarrow
\(\) match u
\(\) One\(x, y, z \) -> \(\) (Unit\(x, y, z \), b)
\(\) Two\(y, z, _ \) ->
\(\) val BCons\(b', _ , _ \) = b
\(\) \(\) More\(Pair\(y, z, Pad\), q, u \), b'
\(\) Three\(y, z, w \) ->
\(\) val BCons\(b', _ , _ \) = b
\(\) \(\) More\(Pair\(y, Empty, One\(z, Pad, Pad\)\), b'
\(\) Unit\(p, _ , _ \) ->
\(\) match p
\(\) Pair\(x, y, _ \) -> \(\) (More\(Pair\(x, y, Pad\), Empty, u \), b)
\(\) Triple\(x, y, z \) ->
\(\) val BCons\(b', _ , _ \) = b
\(\) \(\) More\(Pair\(y, z, Pad\), Pad, Pad\), q, u \), b'
More\(0\(p, q1, u1 \) \) ->
\(\) match p
\(\) Pair\(x, y \) ->
\(\) val \(q1', b'\) = more\(q1, u1 \), b
\(\) \(\) More\(Pair\(x, y, Pad\), q1', u \), b'
\(\) Triple\(x, y, z \) ->
\(\) val BCons\(b', _ , _ \) = b
\(\) \(\) More\(Pair\(y, z, Pad\), Pad, Pad\), q1, u1 \), u \), b'
More\(Pair\(p, y1, q1, u1 \) \) ->
\(\) match p
\(\) Pair\(x, y \) \rightarrow (More\(Pair\(x, y, Pad\), More\(Pair\(y1, q1, u1 \), u \), b)
\(\) Triple\(x, y, z \) ->
\(\) val BCons\(b', _ , _ \) = b
\(\) \(\) More\(Pair\(y, z, Pad\), y1, Pad\), Pad, Pad\), q1, u1 \), u \), b'
More\(Triple\(p, y1, z1, q1, u1 \) \) ->
\(\) val BCons\(b', _ , _ \) = b
\(\) match p
\(\) Pair\(x, y \) -> (More\(Pair\(x, y, Pad\), More\(Pair\(y1, z1, Pad\), Pad, q1, u1 \), u \), b'
\(\) Triple\(x, y, z \) ->
\(\) (More\(Pair\(y, z, Pad\), y1, z1\), Pad, Pad\), y1, z1\), q1, u1 \), u \), b'

fip fun bsnoc\(bs : \) bseq\(\langle a \rangle\), \(u3 : \) reuse3, \(x : \) a\) : exn bseq\(\langle a \rangle\)
val BSeq\(s, b = bs\)
val \(s', b'\) = snoc\(s, b, u3, x\)
BSeq\(s', b'\)
B.1 Properties of Stores

In the following, we will often need to focus on a specific part of the store that corresponds to all values reachable from a root set \( \Gamma \). For a linear store with \( \Gamma \subseteq \text{roots}(S) \), we write \( S[\Gamma] \) for the smallest linear subset of \( S \) containing \( \Gamma \). Then we can split any store with roots \( \Gamma_1, \Gamma_2 \) into \( S[\Gamma_1], S[\Gamma_2] \) and \( S_{\text{cyc}} \) where \( S_{\text{cyc}} \) is the largest linear subset of \( S \) with no roots.

**Lemma 1.** (Store splitting)

Let \( S \) be a linear store, \( \Gamma_1 \cap \Gamma_2 = \emptyset \) and \( \Gamma_1, \Gamma_2 = \text{roots}(S) \). Then \( S[\Gamma_1], S[\Gamma_2] \) and \( S_{\text{cyc}} \) are pairwise disjoint linear stores and \( S = S[\Gamma_1], S[\Gamma_2], S_{\text{cyc}} \).

**Proof.** We first show that they are pairwise disjoint: We use lemma 2 (see below) to prove that for any \( x \in \text{dom}(S[\Gamma_1]) \), we have \( x \notin \text{dom}(S[\Gamma_2]) \) and \( x \notin \text{dom}(S_{\text{cyc}}) \). By symmetry, the same holds with \( \Gamma_1 \) and \( \Gamma_2 \) swapped which implies pairwise disjointness.
We call this property cycle-free. We can combine any linear stores again to obtain a linear store: $P$ we have

The above lemma makes use of the fact that $S$ is linear, this implies that $S$ is a linear store with $\operatorname{dom} S$ being the smallest such set, this implies $S'$ is linear: clearly, since $S$ is linear, so is any subset of $S$. But since $S$ is linear, this implies that $S$ is linear and $\Gamma$ is a linear store with $\operatorname{dom} S$ being the smallest such set, this implies $S'$ is linear.

The above lemma makes use of the fact that $S[\Gamma]$ contains just values reachable from the roots $\Gamma$. We call this property cycle-free. Formally, a store $S$ is cycle free if for any property $P$ with

- $P(x)$ for all $x \in \Gamma$
- $P(x)$ for all $x \in \operatorname{fv}(v)$ with $y \mapsto v \in S[\Gamma]$, $P(y)$

we have $P(x)$ for all $x \in \operatorname{dom}(S)$. In particular, the induced store is cycle-free:

**Lemma 2. (Reachability from roots)**

Let $S$ be a linear store, $\Gamma \subseteq \operatorname{roots}(S)$ and $S[\Gamma]$ the smallest linear subset of $S$ containing $\Gamma$. Then $S[\Gamma]$ is cycle-free.

**Proof.** Let $S'$ be the subset of $S[\Gamma]$ for which $P$ is not true. We claim that $S[\Gamma] - S'$ is still a linear subset of $\Gamma$ containing $\Gamma$. Since $S[\Gamma]$ being the smallest such set, this implies $S' = \emptyset$.

**Case** Soundness of $S[\Gamma] - S'$: Assume that there is $x \in \operatorname{fv}(\operatorname{rng}(S[\Gamma] - S'))$ with $x \notin \operatorname{dom}(S[\Gamma] - S')$. Choose $y \mapsto v \in (S[\Gamma] - S')$ with $x \in \operatorname{fv}(v)$. Since $y \in \operatorname{dom}(S[\Gamma] - S')$, we have $P(y)$. But since $S[\Gamma]$ is sound, $x \in \operatorname{dom}(S[\Gamma])$ and thus $x \in \operatorname{dom}(S')$. If $x \in \operatorname{dom}(S')$, $P(x)$ is not true, which is a contradiction to the second rule of $P$.

**Case** Linearity of $S[\Gamma] - S'$: Since $S[\Gamma]$ is linear, so is any subset of $S[\Gamma]$.

**Case** $S[\Gamma] - S'$ contains $\Gamma$: By assumption $P(x)$ is true for all $x \in \Gamma$, so $\Gamma \cap S' = \emptyset$.

We can combine any linear stores again to obtain a linear store:

**Lemma 3. (Store joining)**

Let $S_1$ and $S_2$ be disjoint sound/linear stores with roots $\Gamma_1$ and $\Gamma_2$. Then $S_1, S_2$ is a sound/linear store with roots $\Gamma_1, \Gamma_2$.

**Proof.**

**Case** Soundness: Let $x \in \operatorname{fv}(\operatorname{rng}(S_1, S_2))$. Then either $x \in \operatorname{fv}(\operatorname{rng}(S_1))$ or $x \in \operatorname{fv}(\operatorname{rng}(S_2))$ and by the soundness of $S_1$ and $S_2$, we have $x \in \operatorname{dom}(S_1)$ or $x \in \operatorname{dom}(S_2)$.

**Case** Linearity: Let $x \in \operatorname{dom}(S_1, S_2)$. Since $S_1$ and $S_2$ are disjoint, either $x \in \operatorname{dom}(S_1)$ (exclusive) or $x \in \operatorname{dom}(S_2)$. Without loss of generality, assume $x \in \operatorname{dom}(S_1)$. Then $x \notin \operatorname{dom}(S_2)$ and by soundness of $S_2$, $x \notin \operatorname{fv}(\operatorname{rng}(S_2))$. By linearity of $S_1$, $x$ occurs at most once in the free variables of
rng(\(S_1\)). Since it does not occur in the free variables of rng(\(S_2\)), it also occurs at most once in the free variables of rng(\(S_1, S_2\)).

**Case** Roots: Since \(S_1\) is sound and disjoint from \(S_2\), we have \(\Gamma_2 \cap \text{fv}(\text{rng}(S_1)) = \emptyset\). By symmetry, \(\Gamma_1, \Gamma_2 \subseteq \text{roots}(S_1, S_2)\). Assume \(x \in \text{roots}(S_1, S_2)\). Then \(x \in \text{roots}(S_1)\) or \(x \in \text{roots}(S_2)\) and so \(x \in \Gamma_1, \Gamma_2\).

### B.2 Simple Invariant

Our simple invariant maintains that the store remains "well-formed" during evaluation. We use two stores: a "borrowed" store \(S_1\) which is unchanged by evaluation and an "owned" store which can be changed. Our simple invariant \(I(e, \Delta, \Gamma, S_1, S_2)\) is defined as:

- \(\Delta \mid \Gamma \vdash e\)
- \(S_1, S_2\) are disjoint stores
- \(\Delta \subseteq \text{dom}(S_1)\) and \(S_1\) sound
- \(\Gamma = \text{roots}(S_2)\) and \(S_2\) linear

That invariant makes it safe to modify \(S_2\), as all values that the store semantics destroys are in \(\Gamma\) and not used anywhere else in the store (\(\Gamma = \text{roots}(S_2)\)). It would be enough for soundness to demand \(\Gamma \subseteq \text{roots}(S_2)\). However, the stronger assertion directly gives us the garbage-free theorem and by the weakening lemma 4 we can always add separated memory to the store later:

**Lemma 4.** *(Weakening for store semantics)*
If \(S \mid e \rightarrow^* S'\), then \(S, S_1 \rightarrow^* S', S_1 \mid e'\) for any \(S_1\) with \(\text{dom}(S_1) \cap \text{dom}(S) = \emptyset\).

**Proof.** By straight-forward induction on the judgement \(S \mid e \rightarrow^* S'\).

### B.3 Progress

In this section we want to show that the store semantics can progress if the simple invariant is true and operational semantics can progress. We assume throughout that for any function \(f(y; \bar{x}) = e \in \Sigma\), we have \(\text{fv}(e) \subseteq y, \bar{x}\). This is true if \(\Sigma\) is fully in-place:

**Lemma 5.** *(Free variables of FIP expressions are in \(\Delta, \Gamma\))*
If \(\Delta \mid \Gamma \vdash e\), then \(\text{fv}(e) \subseteq \Delta, \Gamma\).

**Proof.** By induction on the judgement \(\Delta \mid \Gamma \vdash e\).

**Case** \text{var}:
- \(\Delta \mid x \vdash x\) (1), given
- \(\text{fv}(x) \subseteq \{x\}\) (2), definition

**Case** \text{atom}:
- \(\Delta \mid \emptyset \vdash C\) (1), given
- \(\text{fv}(C) = \emptyset\) (2), definition

**Case** \text{tuple}:
- \(\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash (v_1, \ldots, v_n)\) (1), given
- \(\Delta \mid \Gamma_1 \vdash v_i\) (2), by \text{tuple}
- \(\text{fv}(v_i) \subseteq \Delta, \Gamma_1\) (3), inductive hypothesis
- \(\text{fv}((v_1, \ldots, v_n)) = \text{fv}(v_1), \ldots, \text{fv}(v_n) \subseteq \Delta, \Gamma_1, \ldots, \Gamma_n\) (4), definition

**Case** \text{reuse}:
\[ \Delta | \Gamma_1, \ldots, Gk \vdash C^k v_1 \ldots v_k \]  
(1), given

\[ \Delta | \Gamma_1 \vdash v_i \]  
(2), by \textsc{reuse}

\[ \text{fv}(v_i) \subseteq \Delta, \Gamma_i \]  
(3), inductive hypothesis

\[ \text{fv}(C^k v_1 \ldots v_k) = \text{fv}(v_1), \ldots, \text{fv}(v_k) \subseteq \Delta, \Gamma_1, \ldots, Gk \]  
(4), definition

\textbf{Case call:}

\[ \Delta | \Gamma \vdash f(\overline{y}; e) \]  
(1), given

\[ \overline{y} \in \Delta, \text{dom}(\Sigma) \]  
(2), by \textsc{call}

\[ \Delta | \Gamma \vdash e \]  
(3), by \textsc{call}

\[ \text{fv}(e) \subseteq \Delta, \Gamma \]  
(4), inductive hypothesis

\[ \text{fv}(f(\overline{y}; e)) = \overline{y}, \text{fv}(e) \subseteq \Delta, \Gamma \]  
(4), definition and \( \text{dom}(\Sigma) \cap \text{fv}(e) = \emptyset \)

\textbf{Case \textsc{bapp}:}

\[ \Delta | \Gamma \vdash y e \]  
(1), given

\[ y \in \Delta \]  
(2), by \textsc{bapp}

\[ \Delta | \Gamma \vdash e \]  
(3), by \textsc{bapp}

\[ \text{fv}(e) \subseteq \Delta, \Gamma \]  
(4), inductive hypothesis

\[ \text{fv}(y e) = y, \text{fv}(e) \subseteq \Delta, \Gamma \]  
(4), definition

\textbf{Case \textsc{empty}:}

\[ \Delta | \Gamma, \phi_0 \vdash e \]  
(1), given

\[ \Delta | \Gamma \vdash e \]  
(2), by \textsc{empty}

\[ \text{fv}(e) \subseteq \Delta, \Gamma \]  
(3), inductive hypothesis

\textbf{Case let:}

\[ \Delta | \Gamma_1, \Gamma_2, \Gamma_3 \vdash \text{let } \overline{x} = e_1 \text{ in } e_2 \]  
(1), given

\[ \Delta, \Gamma_2 | \Gamma_1 \vdash e_1 \]  
(2), by \textsc{let}

\[ \Delta | \Gamma_2, \Gamma_3 \vdash e_2 \]  
(3), by \textsc{let}

\[ \text{fv}(e_1) \subseteq \Delta, \Gamma_2, \Gamma_1 \]  
(4), inductive hypothesis

\[ \text{fv}(e_2) \subseteq \Delta, \Gamma_2, \Gamma_3, \overline{x} \]  
(5), inductive hypothesis

\[ \text{fv(} \text{let } \overline{x} = e_1 \text{ in } e_2 \text{)} = \text{fv}(e_1), \text{fv}(e_2) - \overline{x} \subseteq \Delta, \Gamma_1, \Gamma_2, \Gamma_3 \]  
(6), definition

\textbf{Case \textsc{bmatch}:}

\[ \Delta | \Gamma \vdash \text{match } y \{ C_i \overline{x}_i \mapsto e_i \} \]  
(1), given

\[ y \in \Delta \]  
(2), by \textsc{bmatch}

\[ \Delta, \overline{x}_i | \Gamma \vdash e_i \]  
(3), by \textsc{bmatch}

\[ \text{fv}(e_i) \subseteq \Delta, \overline{x}_i, \Gamma \]  
(4), inductive hypothesis

\[ \text{fv(} \text{match } y \{ C_i \overline{x}_i \mapsto e_i \} \text{)} = y, \text{fv}(e_1) - \overline{x}_1, \ldots, \text{fv}(e_n) - \overline{x}_n \subseteq \Delta, \Gamma \]  
(5), definition

\textbf{Case \textsc{dmatch}!:}

\[ \Delta | \Gamma, x \vdash \text{match! } x \{ C_i \overline{x}_i \mapsto e_i \} \]  
(1), given

\[ \Delta | \Gamma, \overline{x}_i, \phi_k \vdash e_i \]  
(2), by \textsc{dmatch!}

\[ \text{fv}(e_i) \subseteq \Delta, \Gamma, \overline{x}_i \]  
(3), inductive hypothesis

\[ \text{fv(} \text{match! } x \{ C_i \overline{x}_i \mapsto e_i \} \text{)} = x, \text{fv}(e_1) - \overline{x}_1, \ldots, \text{fv}(e_n) - \overline{x}_n \subseteq \Delta, \Gamma \]  
(4), definition

We define \([S - \overline{x}]e\) as the substitution which replaces every variable \(y \in \text{fv}(e) - \overline{x}\) by \([S]y\). Then:

**Lemma 6. (Store Substitution on unused variables)**

If \(x \notin \text{fv}(e)\) or \(x \notin \text{dom}(\Sigma)\), then \([S]e = [S - x]e\).

**Proof.** By induction on \(e\).

**Case Variable:** \( y \neq x \)

\([S - x]y = [S]y\)  
(1), by definition
FP²: Fully in-Place Functional Programming

**Case** Variable: \( x \)

\( x \in \text{fv}(e) \)  
(1), since case necessary
\( x \notin \text{dom}(S) \)  
(2), by (1)
\( [S]x = x \)  
(3), by (2)
\( [S - x]x = x \)  
(4), by definition

**Lemma 7.** (Store Substitution commutes)
If \( S \) sound and \( [S]\overline{x} = \overline{v} \), then \( [S](e[\overline{x}:=\overline{v}]) = ([S - \overline{x}]e)[\overline{x}:=\overline{v}] \).

**Proof.**
\[ [S]\overline{x} = \overline{v} = [S]\overline{v} \]  
(1), store substitution is idempotent
\[ [S](e[\overline{x}:=\overline{v}]) = [S](e[\overline{x}:=\overline{v}]) \]  
(2), by (1)
\[ \text{fv}(\overline{v}) \cap \text{dom}(S) = \emptyset \]  
(3), definition
\[ \text{fv}(\overline{v}) \subseteq \overline{x} \]  
(4), since \( S \) is sound
\[ \overline{x} \cap \text{fv}(e[\overline{x}:=\overline{v}]) \subseteq \text{fv}(\overline{v}) \]  
(5), since \( \overline{x} \) is substituted
\[ [S](e[\overline{x}:=\overline{v}]) = [S - \overline{x}](e[\overline{x}:=\overline{v}]) \]  
(6), lemma 6

**Lemma 8.** (Store semantics reads values)
If \( I(\overline{v}, \Delta, \Gamma, S_1, S_2) \) then \( S_1, S_2 \models \overline{v} \rightarrow_s S_1, S_2' \models \overline{x} \) with \( [S_2]\overline{v} = [S_2']\overline{x} \) and all names in \( \text{dom}(S_2') \) – \( \text{dom}(S_2) \) are fresh.

**Proof.** Using the \((x_1, \ldots, \Box, \ldots, v_n)\) context, view each value \( v \) individually. By induction on \( v \).

**Case** \( x: \) clear
\( S_1, S_2 \models x \rightarrow_s x \)  
(1), reflexivity
\( x \in \text{dom}(S_2) \)  
(2), by invariant

**Case** \( C: \)
\( S_1, S_2 \models C \rightarrow_s S_1, S_2, x \rightarrow C \models x \)  
(1), (atom), fresh \( x \)
\( [S_2, x \rightarrow C] \models x \models [S_2] C \)  
(2), obvious

**Case** \( C^k v_1 \ldots v_k: \)
\[ \Delta \models \Gamma_1, \ldots, \Gamma_k, \phi_k \vdash C^k v_1 \ldots v_k \]  
(1), by reuse
\[ \Delta \models \Gamma_1, \ldots, \Gamma_k \vdash (v_1, \ldots, v_k) \]  
(2), by tuple
\[ I((v_1, \ldots, v_k), \Delta, (\Gamma_1, \ldots, \Gamma_k), S_1, S_2 - \phi_k) \]  
(3), by (2)
\[ S_1, (S_2 - \phi_k) \models (v_1, \ldots, v_k) \rightarrow_s^* S_1, S_2' \models \overline{x} \]  
(4), inductive hypothesis
\[ [S_2']\overline{x} = [S_2 - \phi_k](v_1, \ldots, v_k) \]  
(5), inductive hypothesis
\[ S_1, S_2' \models (v_1, \ldots, v_k) \rightarrow_s^* S_1, S_2', \phi_k \models \overline{x} \]  
(6), weakening (4)
\[ S_1, S_2', \phi_k \models C^k \overline{x} \rightarrow_s^* S_1, S_2', x \rightarrow C^k \overline{x} \models x \]  
(7), (reuse), fresh \( x \)
\[ [S_2'] \models C^k \overline{x} \models [S_2](C^k v_1 \ldots v_k) \]  
(8), by (5) and \( x \) fresh

We write drop \( \overline{x} \); \( e \) as a short-hand for drop \( x_1; \ldots; \text{drop} x_n; \ e \).

**Lemma 9.** (Dropping can progress)
If \( I((\text{drop } \overline{x}; e), \Delta, \Gamma, S_1, S_2) \) then \( S_1, S_2 \models \text{drop } \overline{x}; e \rightarrow_s^* S_1, S_2' \models e \) with \([S_1, S_2] e = [S_1, S_2'] e \).

**Proof.** By induction on \( [S_2] \).

**Case** \( S_2 = \emptyset: \)
\[ \Delta \models \emptyset \vdash \text{drop } \overline{x}; e \]  
(1), since roots(\( S_2 \)) = \( \emptyset \)
\[ \overline{x} = \emptyset \]  
(2), by drop
\[ S_1, S_2 \models \text{drop } \emptyset; e \rightarrow_s^* S_1, S_2 \models e \]  
(3), by reflexivity
Case $S_2 \neq \emptyset$. Let $\overline{x} = x, \overline{x}'$.

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

1, by assumption

$x \notin \Gamma$

2, by (1) and the multiplicity of $x$ in $\Gamma, \overline{x}$.

$x \notin \text{fv}((\text{drop } \overline{x}'; e))$

3, by (2) and lemma 5

$S_2 = S'_2, x \mapsto C^k \overline{y}$

4, by assumption

$[S_1, S_2]e = [S_1, S'_2]e$

5, by (3)

$[S_1, S_2, x] \mapsto C^k \overline{y} \mid \text{drop } x; \text{drop } \overline{x}'; e \rightarrow_s S_1, S'_2 \mid \text{drop } (\overline{y}, \overline{x}'); e$

6, by (decon)

$[S'_2] + 1 = [S_2]$

7, by definition

$\Delta \mid \Gamma, \overline{y}, \overline{x}' \vdash \text{drop } (\overline{y}, \overline{x}'); e$

8, by (1) and drop

$\Gamma, x, \overline{x}' = \text{roots}(S_2)$

9, by assumption

$\Gamma, \overline{y}, \overline{x}' = \text{roots}(S'_2)$

10, by (4)

$I((\text{drop } (\overline{y}, \overline{x}')); e), \Delta, (\Gamma, (\overline{y}, \overline{x}'), S_1, S_2)$

11, by (8) and (10)

$S_1, S'_2 \mid \text{drop } (\overline{y}, \overline{x}'); e \rightarrow_{s'} S_1, S''_2 \mid e$

12, by inductive hypothesis

$[S_1, S'_2]e = [S_1, S''_2]e$

13, by inductive hypothesis

$[S_1, S_2]e = [S_1, S'_2]e$

14, by (5) and (13)

Since free variables and static functions are separate syntactic categories, we define store substitution $[S]e$ to not act on functions. This assumption simplifies our proof, but does not actually change its semantics, since by lemma 36, we know that the store never contains functions anyway. Then we can take steps in the store semantics yielding the same value as the operational semantics:

**Lemma 10.** *(Store semantics can progress (no eval ctx))*

If $I(e, \Delta, \Gamma, S_1, S_2)$ and $[S_1, S_2]e \rightarrow e'$, then $S_1, S_2 \mid e \rightarrow_{s'} S_1, S'_2 \mid e''$ with $e' = [S_1, S'_2]e''$.

**Proof.** By case-analysis on $[S_1, S_2]e \rightarrow e'$.

**Case (let):**

$[S_1, S_2](\text{let } \overline{x} = \overline{v} \text{ in } e') = (\text{let } \overline{x} = [S_1, S_2]\overline{v} \text{ in } [S_1, S_2 - \overline{x}]e')$

1, definition

$\rightarrow ([S_1, S_2 - \overline{x}]e')[\overline{x} := ([S_1, S_2]\overline{v})]$

2, by (let)

$S_2 = S_2[\Gamma_1], S_2[\Gamma_2], S_2[\Gamma_3], S_{\text{cyc}}$

3, by assumption

$I(\overline{v}, \emptyset, \Gamma_1, \emptyset, S_2[\Gamma_1])$

4, since $\emptyset \mid \Gamma_1 \vdash \overline{v}$

$I(\overline{v}, \emptyset, \Gamma_1, (S_1, S_2[\Gamma_2], S_2[\Gamma_3], S_{\text{cyc}}), S_2[\Gamma_1])$

5, weakening (4)

$S_1, S_2 \mid \overline{v} \rightarrow_{s'} S_1, S'_2 \mid \overline{v}$

6, lemma 8

$S'_2 = S'_2[\Gamma_1], S_2[\Gamma_2], S_2[\Gamma_3], S_{\text{cyc}}$

7, by (5)

$[S_1, S'_2]\overline{v} = [S_1, S_2]\overline{v}$

8, lemma 8

$[S_1, S'_2 - \overline{x}]e' = [S_1, S_2 - \overline{x}]e'$

9, since all new bindings are fresh

$S_1, S'_2 \mid \text{let } \overline{x} = \overline{z} \text{ in } e' \rightarrow_{s'} S_1, S''_2 \mid e'[\overline{x} := \overline{z}]$

10, (let$_s$)

$[S_1, S''_2][e'[\overline{x} := \overline{z}]] = ([S_1, S'_2 - \overline{x}]e')[\overline{x} := ([S_1, S'_2]\overline{z})]$

11, by lemma 7

$([S_1, S'_2 - \overline{x}]e')[\overline{x} := ([S_1, S'_2]\overline{z})] = ([S_1, S_2 - \overline{x}]e')[\overline{x} := ([S_1, S_2]\overline{z})]$

12, by (8) and (9)

**Case (call):**
FP²: Fully in-Place Functional Programming

\[ [S_1, S_2](f(\overline{y}; \overline{v})) = f([S_1, S_2]\overline{y}; [S_1, S_2]\overline{v}) \]
\[ \rightarrow e[\overline{y}:=([S_1, S_2]\overline{y}), \overline{x}:=([S_1, S_2]\overline{v})] \]  
(1), definition

\[ I(f(\overline{y}; \overline{v})), \Delta, \Gamma, S_1, S_2) \]
\[ \rightarrow \]  
(2), by \( \text{call} \) with \( f(\overline{y}; \overline{x}) = e \in \Sigma \)

\[ I(\overline{v}, \Delta, \Gamma, S_1, S_2) \]
\[ \rightarrow \]  
(3), by assumption

\[ S_1, S_2 | \overline{v} \rightarrow^s \overline{y}, S_1', S_2' | \overline{z} \]
\[ [S_1, S_2'][\overline{z}] = [S_1, S_2][\overline{v}] \]
\[ [S_1, S_2'][\overline{y}'] = [S_1, S_2][\overline{y}] \]
\[ S_1, S_2' | f(\overline{y'}; \overline{z}) \rightarrow_s S_1, S_2' | e[\overline{y'}:=\overline{y}, \overline{x}:=\overline{z}] \]
\[ [S_1, S_2'](e[\overline{y'}:=\overline{y}, \overline{x}:=\overline{z}]) \]
\[ = \]  
(7), since all new bindings are fresh

\[ \text{Case (app)}: \]
\[ [S_1, S_2](f(\overline{v})) = (f)([S_1, S_2]\overline{v}) \]
\[ \rightarrow e[\overline{x}:=([S_1, S_2]\overline{v})] \]
\[ I(f(\overline{v}), \Delta, \Gamma, S_1, S_2) \]
\[ \rightarrow \]  
(1), by \ref{app}

\[ I(\overline{v}, \Delta, \Gamma, S_1, S_2) \]
\[ \rightarrow \]  
(2), by \( \text{app} \) with \( f(; \overline{x}) = e \in \Sigma \)

\[ S_1, S_2 | \overline{v} \rightarrow^s \overline{y}, S_1', S_2' | \overline{z} \]
\[ [S_1, S_2'][\overline{z}] = [S_1, S_2][\overline{v}] \]
\[ S_1, S_2' | (f \overline{z}) \rightarrow_s S_1, S_2' | e[\overline{z}:=\overline{z}] \]
\[ [S_1, S_2'](e[\overline{z}:=\overline{z}]) \]
\[ = \]  
(8), by \( \text{call}_1 \) with \( f(\overline{y}; \overline{x}) = e \in \Sigma \)

\[ \text{Case (match)}: \]
\[ [S_1, S_2](\text{match } y \{ \overline{p} \mapsto e \}) \]
\[ = (\text{match } (C \overline{v}) \{ \overline{p} \mapsto [S_1, S_2]e \}) \]
\[ \rightarrow ([S_1, S_2 - \overline{y}]e_1)[\overline{y}:=\overline{v}] \]
\[ I((\text{match } y \{ \overline{p} \mapsto e \}), \Delta, \Gamma, S_1, S_2) \]
\[ \rightarrow \]  
(1), definition, where \( [S_1, S_2]y = C^k \overline{v} \)

\[ y \mapsto C^k \overline{z} \in S_1 \]
\[ [S_1, S_2]y = [S_1, S_2](C^k \overline{z}) = C^k \overline{v} \]
\[ S, y \mapsto C^k \overline{z} | \text{match } y \{ \overline{p} \mapsto e \} \rightarrow_s S, y \mapsto C^k \overline{z} | e_1[\overline{y}:=\overline{z}] \]
\[ [S_1, S_2](e_1[\overline{y}:=\overline{z}]) \]
\[ = \]  
(5), since \( [S_1, S_2]y = C^k \overline{v} \)

\[ \text{Case (match!)}: \]
\[ [S_1, S_2](\text{match! } x \{ \overline{p} \mapsto e \}) \]
\[ = (\text{match! } (C \overline{v}) \{ \overline{p} \mapsto [S_1, S_2]e \}) \]
\[ \rightarrow ([S_1, S_2 - \overline{x}]e_1)[\overline{x}:=\overline{v}] \]
\[ I((\text{match! } x \{ \overline{p} \mapsto e \}), \Delta, \Gamma, S_1, S_2) \]
\[ \rightarrow \]  
(1), definition, where \( [S_1, S_2]x = C^k \overline{v} \)

\[ x \mapsto C^k \overline{z} \in S_2 \]
\[ [S_1, S_2]x = [S_1, S_2](C^k \overline{z}) = C^k \overline{v} \]
\[ S, x \mapsto C^k \overline{z} | \text{match! } x \{ \overline{p} \mapsto e \} \rightarrow_s S, x \mapsto C^k \overline{z} | e_1[\overline{x}:=\overline{z}] \]
\[ [S_1, S_2](e_1[\overline{x}:=\overline{z}]) \]
\[ = \]  
(5), since \( [S_1, S_2]x = C^k \overline{v} \)

\[ \text{Case (drop)}: \]
\[ I((\text{drop } \overline{x}; e), \Delta, \Gamma, S_1, S_2) \]
\[ \rightarrow \]  
(1), given

\[ S_1, S_2 | \text{drop } \overline{x}; e \rightarrow^s S_1, S_2' | e \]
\[ [S_1, S_2]e = [S_1, S_2']e \]
\[ (2), by \ref{drop} \]

\[ (3), by \ref{drop} \]
Case (free):

$I((\text{free } k; e), \Delta, \Gamma, S_1, S_2)\quad (1)$, given

$\circ_k \in S_2\quad (2)$, given

$S_1, S_2, \circ_k \mid \text{free } k; e \rightarrow_s S_1, S_2 \mid e\quad (3)$, by (free)

$[S_1, S_2, \circ_k]e = [S_1, S_2]e\quad (4)$, obvious

B.4 Store semantics preserves linearity and roots

In this section, we wish to maintain the invariant $I(e, \Delta, \Gamma, S_1, S_2)$ which makes lemma 10 work.

Lemma 11. (Variable substitution preserves FIP typing)

Assuming that $\bar{y}$ does not occur in $e$. (1) If $\Delta \mid \Gamma, \bar{x} \vdash e$, then $\Delta \mid \Gamma, \bar{y} \vdash e[\bar{x}:=\bar{y}]$. (2) If $\Delta, \bar{x} \mid \Gamma \vdash e$, then $\Delta, \bar{y} \mid \Gamma \vdash e[\bar{x}:=\bar{y}]$.

Proof. By induction on the FIP judgement for any such $\bar{x}, \bar{y}$.

Case var:

$\Delta \mid x \vdash x$ given

$\Delta \mid y \vdash x[x:=y]$ given

Case let:

$\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3, \bar{x}_1, \bar{x}_2, \bar{x}_3 \vdash e_1$ in $e_2$ given

$\Delta \mid \Gamma_2, \Gamma_3, \bar{x}_2, \bar{x}_3, \bar{z} \vdash e_2$ given

$\Delta \mid \Gamma_2, \Gamma_3, \bar{y}_2, \bar{y}_3, \bar{z} \vdash e_2[\bar{x}_2:=\bar{y}_2, \bar{x}_3:=\bar{y}_3]$, inductive hypothesis (1)

$\Delta, \Gamma_2, \bar{x}_2 \mid \Gamma_1, \bar{x}_1 \vdash e_1$ given

$\Delta, \Gamma_2, \bar{x}_2 \mid \Gamma_1, \bar{y}_1 \vdash e_1[\bar{x}_1:=\bar{y}_1]$, inductive hypothesis (1)

$\Delta, \Gamma_2, \bar{y}_2 \mid \Gamma_1, \bar{y}_1 \vdash e_1[\bar{x}_1:=\bar{y}_1, \bar{x}_2:=\bar{y}_2]$, inductive hypothesis (2)

$\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3, \bar{y}_1, \bar{y}_2, \bar{y}_3 \vdash (\text{let } \bar{z} = e_1 \text{ in } e_2)[\bar{x}:=\bar{y}]$ above

Other cases are clear.

Lemma 12. (The delta environment can be weakened)

If $\Delta \mid \Gamma \vdash e$, then $\Delta, x \mid \Gamma \vdash e$ for any $x \notin \Gamma$.

Proof. By straight-forward induction on the judgement $\Delta \mid \Gamma \vdash e$. We have to require $x \notin \Gamma$ as we have to ensure $\Delta, x \cap \Gamma = \emptyset$.

The next lemma is the main lemma of our soundness proof. It says that the individual steps of the store semantics take sound/linear stores to sound/linear stores. As usual, the judgement $\rightarrow_s$ does not include the eval rule.

Lemma 13. (Store semantics preserves linearity and roots (no eval ctx))

If $I(e, \Delta, \Gamma, S_1, S_2)$ and $S_1, S_2 \mid e \rightarrow_s S_3$, then $I(e', \Delta', \Gamma', S_1, S_3)$.

Proof. By case analysis on the rules of the store semantics.

Case (let$_s$).

$S_3 := S_2$ define

$\Delta \mid \Gamma, \bar{y} \vdash \bar{z}$ in $e$ given

$\Gamma, \bar{y} = \text{roots}(S_2) = \text{roots}(S_3)$ given

$\Delta \mid \Gamma, \bar{x} \vdash e$ by let

$\Delta \mid \Gamma, \bar{y} \vdash e[\bar{x}:=\bar{y}]$ by lemma 11

Case (call$_s$).
FP²: Fully in-Place Functional Programming

\[ S_k := S_2 \]

define

\[ \Delta, \bar{y}' \mid \bar{x}' \vdash f(\bar{y}'; \bar{x}') \]
given

\[ \bar{x}' = \text{roots}(S_2) = \text{roots}(S_3) \]
given

\[ \bar{y}' \subseteq \text{dom}(S_1) \]
given

\[ \bar{y}' \mid \bar{x}' \vdash e \]
by \text{DEFFUN} and \( f(\bar{y}; \bar{x}) = e \in \Sigma \)

\[ \bar{y}' \mid \bar{x}' \vdash e[\bar{x}:=\bar{x}', \bar{y}:=\bar{y}'] \]
by lemma 11

Case (\textit{anon}_x).

\[ S_k := S_2 \]

(1), define

\[ \Delta, f \mid \bar{x}' \vdash f \bar{x}' \]
(2), given

\[ \bar{x}' = \text{roots}(S_2) = \text{roots}(S_3) \]
(3), by (2)

\[ \emptyset \mid \bar{x}' \vdash e \]
(4), by \text{DEFFUN} and \( f(\bar{ }; \bar{x}) = e \in \Sigma \)

\[ \emptyset \mid \bar{x}' \vdash e[\bar{x}:=\bar{x}'] \]
(5), by lemma 11

Case (\textit{reuse}_x).

\[ \Delta \mid \emptyset \vdash C_k \bar{x} \]
(1), given

\[ \emptyset_k \in \text{roots}(S_2) \]
(2), by (1)

\[ S_k := S_2 - \emptyset_k, x \mapsto C_k \bar{x} \]
(3), define, well-defined by (2)

\[ \Delta \mid \emptyset \vdash x \]
(4), by \text{VAR}

\[ \bar{x}, \emptyset_k = \text{roots}(S_2) \]
(5), by (1)

\[ x = \text{roots}(S_3) \]
(6), by (3),(5) and since \( x \) is fresh

\[ S_3 \text{ linear} \]
(7), by (6)

Case (\textit{atom}_x).

\[ \Delta \mid \emptyset \vdash C \]
(1), given

\[ S_2 = \emptyset \]
(2), by (1)

\[ S_3 := x \mapsto C \]
(3), define

\[ \Delta \mid \emptyset \vdash x \]
(4), by \text{VAR}

\[ x' = \text{roots}(S_3) \]
(5), by (3)

\[ S_3 \text{ linear} \]
(6), since \( x \) is fresh

Case (\textit{bmatch}_x).

\[ \Delta \mid \Gamma \vdash \text{match} y \{ \bar{p} \mapsto e \} \]
(1), given

\[ y \mapsto C_k \bar{y} \in \text{dom}(S_1) \]
(2), given

\[ \bar{y} \subseteq \text{dom}(S_1) \]
(3), since \( S_1 \) is sound

\[ \Delta, \bar{y} \mid \Gamma \vdash e[\text{bv}(p):=\bar{y}] \]
(4), by (1) and 11

\[ S_3 := S_2 \]
(5), define

Case (\textit{dmatch}_x).

\[ S_3 := S_2 - x, \emptyset_k \]
(1), define

\[ \Delta \mid \Gamma \vdash \text{match} x \{ \bar{p} \mapsto e \} \]
(2), given

\[ x \mapsto C_k \bar{y} \in \text{roots}(S_2) \]
(3), given

\[ S_3 \text{ sound} \]
(4), by (3)

\[ \bar{y} \subseteq \text{roots}(S_3) \]
(5), since \( S_2 \) is linear

\[ \Delta \mid \Gamma, \bar{y}, \emptyset_k \vdash e[\text{bv}(p):=\bar{y}] \]
(6), by (2) and 11

Case (\textit{dcon}_x).
We now want to generalize lemma 13 to arbitrary evaluation contexts. In particular, we might start with $I(E[e], \Delta, \Gamma, S_1, S_2)$, take a step $S_1, S_2 \mid e \rightarrow S_1, S_3 \mid e'$ and want to obtain $I(E[e'], \Delta', \Gamma', S_1, S_3)$. But unfortunately, this does not work. Let’s see where this breaks down. We define:

1. $E = \text{let } \overline{x} = \overline{y} \text{ in drop } \overline{x}; y$
2. $e = \text{match } y \{ C \overline{y} \mapsto f(\overline{y}); \}$

Then:

- We start with $I(E[e], \Delta, \emptyset, \{y\}, \emptyset, S_2)$
- We have $y \mid \emptyset \vdash e$, so $I(e, \{y\}, \emptyset, S_2, \emptyset)$
- We step $S_2 \mid \text{match } y \{ C \overline{y} \mapsto f(\overline{y}); \} \rightarrow S_2 \mid f(\overline{y};)$
- Now we have $\overline{y} \mid \emptyset \vdash f(\overline{y};)$, so $I(e', \{\overline{y}\}, \emptyset, S_2, \emptyset)$

But then we do not have:

- $I(E[f(\overline{y};)], \emptyset, \{y\}, \emptyset, S_2)$, because the let-rule only allows us to borrow $y$, not $\overline{y}$
- $I(E[f(\overline{y};)], \emptyset, \{y, \overline{y}\}, \emptyset, S_2)$, because $\overline{y}$ are not used as owned.
- $I(E[f(\overline{y};)], \{\overline{y}\}, \emptyset, S_2)$, because clearly $\overline{y} \not\in \emptyset$.

The problem seems to be that the $\Delta$ environment of the nested computation can change in such a way, that we can not put the environment back together in our invariant. Therefore, we need a weaker assumption which is maintained across nested evaluation steps. In this weaker invariant, we maintain the split of the evaluation context explicitly and allow a new $\Delta'$ environment for the contained expression. We define $I_E(e, \Delta, \Gamma, S_1, S_2)$ by induction on $E$:

**Case** $E = \emptyset$:

$I_E(e, \Delta, \Gamma, S_1, S_2) := I(e, \Delta, \Gamma, S_1, S_2)$

**Case** $E[E'] = \bigwedge^k x_1 \ldots E' \ldots v_k \mid (x_1, \ldots, E', \ldots, v_n) \mid E' e \mid x E' \mid f(\overline{y}; E') \mid \text{match } E' \{ \overline{p} \mapsto \overline{e}; \}$: $I_E[E'](e, \Delta, (\Gamma_i, \Gamma), S_1, (S_2[\Gamma_i], S_2')) := I_E(e, \Delta', \Gamma_i, (S_1, S_2'), S_2[\Gamma_i])$ and $I(E[()], \Delta, \Gamma, S_1, S_2')$

**Case** $E[E'] = \text{let } \overline{x} = E' \text{ in } e$:

$I_E[E'](e, \Delta, (\Gamma_i, \Gamma_2, \Gamma_3), S_1, (S_2[\Gamma_i], S_2')) := I_E(e, \Delta', \Gamma_1, (S_1, S_2'), S_2[\Gamma_i])$

and $I(E[()], \Delta, (\Gamma_2, \Gamma_3), S_1, S_2')$

The big difference between this new invariant and the statement $I(E[e], \Delta, \Gamma, S_1, S_2)$, is that we allow an arbitrary $\Delta'$ environment in the recursive case (as long as $\Delta'$ continues to be a subset of $S_1, S_2$). This simply weakens the previous invariant:

**Lemma 14.** *(Weakening the invariant)*

If $I(E[e], \Delta, \Gamma, S_1, S_2)$, then $I_E(e, \Delta, \Gamma, S_1, S_2)$.

By induction on $E$:  

\[ S_3 := S_2 - x \quad (1), \text{ define} \]
\[ \Delta \mid \Gamma, x \vdash \text{drop } x; e \quad (2), \text{ given} \]
\[ x \mapsto C^k \overline{y} \text{ in } \text{roots}(S_2) \quad (3), \text{ given} \]
\[ S_3 \text{ sound} \quad (4), \text{ by (3)} \]
\[ \overline{y} \subseteq \text{roots}(S_3) \quad (5), \text{ since } S_2 \text{ is linear} \]
\[ \Delta \mid \Gamma, \text{drop } \overline{y}; e \quad (6), \text{ by (2), drop} \]
\[E = \square:\]
\[I(e, \Delta, \Gamma, S_1, S_2) \quad (1), \text{ given}\]
\[I_\square(e, \Delta, \Gamma, S_1, S_2) \quad (2), \text{ by definition}\]

**Case** \(E[E'] = C^k x_1 \ldots E' \ldots v_k \mid (x_1, \ldots, E', \ldots, v_n) \mid E' \mid x \ E' \mid f(\overline{y}; E') \mid \text{match } E' \{ \overline{p} \mapsto \overline{e} \} :\)

\[I(E[E'][e], \Delta, \Gamma, S_1, S_2) \quad (1), \text{ given}\]
\[\Gamma = \Gamma_1, \Gamma' \text{ and } S_2 = S_2[\Gamma_1], S'_2 \quad (2), \text{ by the relevant rule}\]
\[I(E'[e], \Delta, \Gamma_2, S_1, S_2[\Gamma_1]) \quad (3), \text{ by (2)}\]
\[I(E[()], \Delta, \Gamma, S_1, S'_2) \quad (4), \text{ by (2)}\]
\[I_{E'}(e, \Delta, \Gamma_1, S_1, S'_2[\Gamma_1]) \quad (5), \text{ inductive hypothesis on (3)}\]
\[I_{E[E']}(e, \Delta, \Gamma, S_1, S_2) \quad (6), \text{ by definition}\]

**Case** \(E[E'] = \text{let } \overline{x} = E' \text{ in } e:\)

\[I(E[E'][e], \Delta, \Gamma, S_1, S_2) \quad (1), \text{ given}\]
\[\Gamma = \Gamma_1, \Gamma_2, \Gamma_3 \text{ and } S_2 = S_2[\Gamma_1], S'_2 \quad (2), \text{ by let}\]
\[I(E'[e], \Delta, \Gamma_2, \Gamma_3, (S_1, S'_2), S_2[\Gamma_1]) \quad (3), \text{ by (2)}\]
\[I(E[()], \Delta, \Gamma, S_1, S'_2) \quad (4), \text{ by (2)}\]
\[I_{E'}(e, \Delta, \Gamma_1, S_1, S'_2[\Gamma_1]) \quad (5), \text{ inductive hypothesis on (3)}\]
\[I_{E[E']} (e, \Delta, \Gamma, S_1, S_2) \quad (6), \text{ by definition}\]

Crucially, we can also weaken the simple variant inside the main invariant:

**Lemma 15.** *(Weakening the invariant)*

If \(I_{E[E_1]}(e, \Delta, \Gamma, S_1, S_2), \text{ then } I_{E[E_1]}(e, \Delta, \Gamma, S_1, S_2).\)

By induction on \(E:\)

**Case** \(E = \square:\)

\[I_{\square} (E_2[e], \Delta, \Gamma, S_1, S_2) \quad (1), \text{ given}\]
\[I_{E_2[e]} (e, \Delta, \Gamma, S_1, S_2) \quad (2), \text{ by definition}\]
\[I_{E_2[e]}(e, \Delta, \Gamma, S_1, S_2) \quad (3), \text{ by lemma 14}\]
\[I_{E_2[e]} (e, \Delta, \Gamma, S_1, S_2) \quad (4), \text{ by (3)}\]

**Case** \(E[E'] = C^k x_1 \ldots E' \ldots v_k \mid (x_1, \ldots, E', \ldots, v_n) \mid E' \mid x \ E' \mid f(\overline{y}; E') \mid \text{match } E' \{ \overline{p} \mapsto \overline{e} \} :\)

\[I_{E[E']}(E_2[e], \Delta, (\Gamma_1, \Gamma_3), S_1, (S_2[\Gamma_1], S'_2)) \quad (1), \text{ given}\]
\[I_{E_2[e]}(E_2[e], \Delta', \Gamma_1, S_1, S_2[\Gamma_1]) \quad (2), \text{ by definition}\]
\[I(E[()], \Delta, \Gamma, S_1, S'_2) \quad (3), \text{ by definition}\]
\[I_{E'}(e, \Delta', \Gamma_1, S_1, S'_2[\Gamma_1]) \quad (4), \text{ by inductive hypothesis on (2)}\]
\[I_{E[E_2]}(e, \Delta, (\Gamma_1, \Gamma_3), S_1, (S_2[\Gamma_1], S'_2)) \quad (5), \text{ by (3) and (4)}\]

**Case** \(E[E'] = \text{let } \overline{x} = E' \text{ in } e:\)

\[I_{E[E']}(E_2[e], \Delta, (\Gamma_1, \Gamma_3), \Gamma_3, S_1, (S_2[\Gamma_1], S'_2)) \quad (1), \text{ given}\]
\[I(E_2[e], \Delta', \Gamma_1, S_1, S'_2[\Gamma_1]) \quad (2), \text{ by definition}\]
\[I(E[()], \Delta, (\Gamma_1, \Gamma_3), S_1, S'_2) \quad (3), \text{ by definition}\]
\[I_{E'}(e, \Delta', \Gamma_1, S_1, S'_2[\Gamma_1]) \quad (4), \text{ by inductive hypothesis on (2)}\]
\[I_{E[E_2]}(e, \Delta, (\Gamma_1, \Gamma_3), S_1, (S_2[\Gamma_1], S'_2)) \quad (5), \text{ by (3) and (4)}\]

Of course, we do not want to remain in the weaker invariant forever. But how can we get back, if the \(\Lambda'\) environment is wrong? The trick is that we have to wait until execution of the nested expression is finished. Then, any resulting value can be typed with an empty \(\Lambda'\) environment:

**Lemma 16.** *(Values do not borrow)*

If \(\Delta \mid \Gamma \vdash \overline{x}\), then \(\emptyset \mid \Gamma \vdash \overline{\nu}\).
Proof. By induction on $\Delta \vdash \Gamma \vdash v$:

**Case** $x$: clear

$\Delta \vdash x \vdash x$ (1), given

$\emptyset \vdash x \vdash x$ (2), by var

**Case** $C$:

$\Delta \vdash \emptyset \vdash C$ (1), given

$\emptyset \vdash \emptyset \vdash C$ (2), by atom

**Case** $C^k v_1 \ldots v_k$:

$\Delta \vdash \Gamma_1, \ldots, \Gamma_k, \circ_k \vdash C^k v_1 \ldots v_k$ (1), by reuse

$\Delta \vdash \Gamma_i \vdash v_i$ (2), by (1)

$\emptyset \vdash \Gamma_i \vdash v_i$ (3), inductive hypothesis

$\emptyset \vdash \Gamma_1, \ldots, \Gamma_k, \circ_k \vdash C^k v_1 \ldots v_k$ (4), by reuse

**Case** $(v_1, \ldots, v_k)$:

$\Delta \vdash \Gamma_1, \ldots, \Gamma_k \vdash (v_1, \ldots, v_k)$ (1), by tuple

$\Delta \vdash \Gamma_i \vdash v_i$ (2), by (1)

$\emptyset \vdash \Gamma_i \vdash v_i$ (3), inductive hypothesis

$\emptyset \vdash \Gamma_1, \ldots, \Gamma_k \vdash (v_1, \ldots, v_k)$ (4), by tuple

**Lemma 17.** *(Values do not borrow)*

If $I(\emptyset, \Delta, \Gamma, S_1, S_2)$, then $I(\emptyset, \emptyset, \Gamma, S'_1, S_2)$ for any $S'_1$.

**Proof.**

$I(\emptyset, \Delta, \Gamma, S_1, S_2)$ (1), given

$\Delta \vdash \Gamma \vdash \emptyset$ (2), by definition

$S_1, S_2$ are disjoint stores (3), by definition

$\Delta \subseteq \text{dom}(S_1)$ and $S_1$ sound (4), by definition

$\Gamma = \text{roots}(S_2)$ and $S_2$ linear (5), by definition

$\emptyset \vdash \Gamma \vdash \emptyset$ (6), by lemma 16

$I(\emptyset, \emptyset, \Gamma, S'_1, S_2)$ (7), above

However, notice that we introduce a $\Delta'$ envirement at every level of the evaluation context. That means, that we can only remove a single level of the evaluation context at a time. To model this, we call an evaluation context $E$ flat if $E = E'[E'']$ implies that either $E'$ or $E''$ is the hole. Then:

**Lemma 18.** *(Strengthening the invariant)*

If $E$ is flat and $I_E(\emptyset, \Delta, \Gamma, S_1, S_2)$, then $I(E[\emptyset], \Delta, \Gamma, S_1, S_2)$.

**Proof.** By case analysis on $E$.

**Case** $E = \Box$:

$I_E(\emptyset, \Delta, \Gamma, S_1, S_2)$ (1), given

$I(\emptyset, \Delta, \Gamma, S_1, S_2)$ (2), by definition

**Case** $E = C^k x_1 \ldots \Box \ldots \Box v_k | (x_1, \ldots, \Box, \ldots, v_n) | \Box e | x \Box | \text{f}(\emptyset; \Box) | \text{match} \Box \{ \overrightarrow{p} \rightarrow e \}$:
FP²: Fully in-Place Functional Programming

\[ I_E(\overline{v}, \Delta, (\Gamma_1, \Gamma), S_1, (S_2[\Gamma_1], S'_2)) \]  
(1), given

\[ I_E(\overline{v}, \Delta', \Gamma, S_1, S_2[\Gamma]) \]  
(2), by definition

\[ I(E[\overline{v}]), \Delta, \Gamma, S_1, S_2 \]  
(3), by definition

\[ I(E[\overline{v}], \Delta, \Gamma, S_1, S_2) \]  
(4), by (2)

\[ I(E[\overline{v}, \Delta, (\Gamma_1, \Delta')], S_1, (S_2[\Gamma_1], S'_2)) \]  
(5), by lemma 17

\[ I_E(\overline{v}), \Delta, (\Gamma_1, \Gamma), S_1, (S_2[\Gamma_1], S'_2)) \]  
(6), by (3) and (5)

**Case** \( E = \text{let } \overline{x} = \emptyset \text{ in } e \):

\[ I_E(\overline{v}, \Delta, \Gamma, S_1, S_2) \]  
(1), given

\[ I_E(\overline{v}, \Delta, \Gamma, S_1, S_2) \]  
(2), by definition

\[ I(E_2[\overline{v}], \Delta, \Gamma, S_1, S_2) \]  
(3), by lemma 18

\[ I(E_2[\overline{v}], \Delta, \Gamma, S_1, S_2) \]  
(4), by (3)

**Case** \( E'E' = C^k x_1 \ldots E' \ldots v_k | (x_1, \ldots, E', \ldots, v_n) | E' e | x E' | f(\overline{v}; E') | \text{match } E' \{ \overline{p} \mapsto \overline{e} \} : 

\[ I_E(E'[E_1])(\overline{v}, \Delta, (\Gamma_1, \Gamma), S_1, (S_2[\Gamma_1], S'_2)) \]  
(1), given

\[ I(E[\overline{v}]), \Delta, \Gamma, S_1, S'_2 \]  
(2), by definition

\[ I_E(E[E_1])(\overline{v}, \Delta', \Gamma, S_1, S'_2, S_2[\Gamma]) \]  
(3), by definition

\[ I_E(E_2[\overline{v}], \Delta', \Gamma, S_1, S'_2, S_2[\Gamma]) \]  
(4), inductive hypothesis on (3)

\[ I_E(E'[E_2])(\overline{v}, \Delta, (\Gamma_1, \Gamma), S_1, (S_2[\Gamma_1], S'_2)) \]  
(5), by (2) and (4)

**Case** \( E'E' = \text{let } \overline{x} = \overline{E'} \text{ in } e \):

\[ I_E(E'[E_1])(\overline{v}, \Delta, (\Gamma_1, \Gamma), S_1, (S_2[\Gamma_1], S'_2)) \]  
(1), given

\[ I_E(E[\overline{v}]), \Delta, (\Gamma_2, \Gamma_3), S_1, S'_2 \]  
(2), by definition

\[ I_E(E[E_1])(\overline{v}, \Delta', \Gamma_1, (\Gamma_1, S'_2), S_2[\Gamma]) \]  
(3), by definition

\[ I_E(E_2[\overline{v}], \Delta', \Gamma_1, (\Gamma_1, S'_2), S_2[\Gamma]) \]  
(4), by inductive hypothesis on (3)

\[ I_E(E'[E_2])(\overline{v}, \Delta, (\Gamma_1, \Gamma), S_1, (S_2[\Gamma_1], S'_2)) \]  
(5), by (2) and (4)

**B.6 Soundness**

Now we can show the main soundness theorem. First, we extend the progress and preservation proofs to handle the **step** and **eval** cases. If we evaluate \( e_1 \) under the context \( E_1 \), we need to assume the invariant \( I_E(e_1, \Delta, \Gamma, S_1, S_2) \). But how can we obtain the invariant for this precise \( E_1 \)? The trick is that we do not have to know \( E_1 \), instead we can just assume \( I_E(e_2, \Delta, \Gamma, S_1, S_2) \) for \( E_1[e_1] = E_2[e_2] \):

**Lemma 20. (Comparing evaluation contexts)**

Let \( E_1[e_1] = E_2[e_2] \), then either:

- \( E_1 = E'_1[E''_1] \) with \( E'_1 = E_2 \) and \( e_1 = E''_1[e_1] \)
- \( E_2 = E'_2[E''_2] \) with \( E_2 = E_1 \) and \( e_1 = E''_2[e_2] \)
Proof. By induction on the pair \((E_1, E_2)\):

**Case** \(E_1 = \square\):
\[
e_1 = E_2[e_2] \quad (1), \text{ given}
\]
\[
E_2 = \square[E_2] \quad (2), \text{ obvious}
\]

**Case** \(E_2 = \square\):
\[
e_2 = E_1[e_1] \quad (1), \text{ given}
\]
\[
E_1 = \square[E_1] \quad (2), \text{ obvious}
\]

**Case** Else: \(E_1 = E_1'[E_1'']\), \(E_2 = E_2'[E_2'']\), and \(E_1', E_2'\) are flat.
\[
E_1'[E_1''] = E_2'[E_2''] \quad (1), \text{ given}
\]
\[
E_1' = E_2' \quad (2), \text{ by (1)}
\]
\[
E_1'' = E_3[E_4] \text{ with } E_3 = E_1'' \text{ and } e_2 = E_4[e_1] \quad (3), \text{ inductive hypothesis, wlog case (1) holds}
\]
\[
E_2 = E_1'[E_1''] = E_1'E_3[E_4]\quad (4), \text{ by (3)}
\]
\[
E_2' = E_2'' \quad (5), \text{ by (2) and (3)}
\]

**Lemma 21. (Alignment of invariant)**
If \(e_1 \rightarrow e_1', E_1[e_1] = E_2[e_2]\) and \(I_{E_1}(e_2, \Delta, \Gamma, S_1, S_2)\), then \(I_{E_1}(e_1, \Delta, \Gamma, S_1, S_2)\).

**Proof.** Use lemma 20 to compare \(E_1\) and \(E_2\).

**Case** \(E_1 = E_1'[E_1'']\) with \(E_1' = E_2\) and \(e_2 = E_1'[e_1]\)
\[
I_{E_1}(e_2, \Delta, \Gamma, S_1, S_2) \quad (1), \text{ given}
\]
\[
I_{E_1}[E_1'] = E_1'[E_1''] \quad (2), \text{ by assumption}
\]
\[
I_{E_1}[E_1'](e_1, \Delta, \Gamma, S_1, S_2) \quad (3), \text{ lemma 15}
\]
\[
I_{E_1}(e_1, \Delta, \Gamma, S_1, S_2) \quad (4), \text{ by (3)}
\]

**Case** \(E_2 = E_2'[E_2'']\) with \(E_2' = E_1\) and \(e_1 = E_2'[e_2]\)
\[
I_{E_2}(e_2, \Delta, \Gamma, S_1, S_2) \quad (1), \text{ given}
\]
\[
I_{E_2}[E_1'](e_2, \Delta, \Gamma, S_1, S_2) \quad (2), \text{ by assumption}
\]
\[
e_1 = E_2''[e_2] \rightarrow e_1' \quad (3), \text{ given}
\]

Use case analysis on (3). In every case, either \(E''_2 = \square\) or \(E''_2\) is flat with \(e_2 = \overline{\nu}\).

If \(E''_2 = \square\), then \(E_2 = E_2' = E_1\) and the claim holds by (1).

If \(E''_2\) is flat with \(e_2 = \overline{\nu}\), then
\[
I_{E_2}(E''_2[\overline{\nu}], \Delta, \Gamma, S_1, S_2) \quad (4), \text{ by (2) and lemma 19}
\]
\[
I_{E_1}(e_1, \Delta, \Gamma, S_1, S_2) \quad (5), \text{ by (4)}
\]

**Lemma 22. (Store semantics can progress)**
If \(I_e(e, \Delta, \Gamma, S_1, S_2)\) and \([S_1, S_2]e \rightarrow e'\), then \(S_1, S_2 | e \rightarrow s_1, S_1', S_2' | e''\) with \(e' = [S_1, S_2']e''\).

**Proof.**
\[
I_e(e, \Delta, \Gamma, S_1, S_2) \quad (1), \text{ given}
\]
\[
I(e, \Delta', \Gamma', s', S_1', S_2'), \text{ with } S_1, S_2 = S_1', S_2' \text{ and } S_2 \subseteq S_2 \quad (2), \text{ by (1)}
\]
\[
[S_1, S_2]e = [S_1', S_2']e \quad (3), \text{ by (2)}
\]
\[
S_1', S_2' | e \rightarrow s_1', S_1', S_3' | e'' \quad (4), \text{ by lemma 10}
\]
\[
e' = [S_1', S_2']e'' \quad (5), \text{ by lemma 10}
\]
\[
S_3 := S_3'(S_2 - S_2') \quad (6), \text{ define}
\]
\[
S_1, S_2 | e \rightarrow s_1, S_1, S_3 | e'' \quad (7), \text{ by (4)}
\]
\[
e' = [S_1, S_3]e'' \quad (8), \text{ by (5)}
\]
Lemma 23. (Store semantics preserves linearity and roots)
If \( I_E(e, \Delta, \Gamma, S_1, S_2) \) and \( S_1, S_2 \models e \longrightarrow_s S_1, S_3 \mid e' \), then \( I_E(e', \Delta', \Gamma', S_1, S_3) \).

**Proof.**

\[
\begin{align*}
I_E(e, \Delta, (\Gamma, \Gamma'), S_1, (S_2, S_2')) & \quad \text{(1), given} \\
I_e(e', \Delta', \Gamma, (S_1, S_2')) & \quad \text{(2), split (1)} \\
I_E((e, \Delta, \Gamma', S_1, S_2') & \quad \text{(3), split (1)} \\
S_1, S_2 | e \longrightarrow_s S_1, S_2', S_3 | e' & \quad \text{(4), by assumption} \\
I_e(e', \Delta'', \Gamma'', (S_1, S_2'), S_3) & \quad \text{(5), by lemma 13} \\
S_3 := S_1', S_2' & \quad \text{(6), define} \\
I_E(e', \Delta, (\Gamma'', \Gamma'), S_1, S_3) & \quad \text{(7), merge (3) and (5)}
\end{align*}
\]

Lemma 24. (Soundness lemma)
If \( I_E(e, \Delta, \Gamma, S_1, S_2) \) and \( [S_1, S_2](E[e]) \longrightarrow^* \bar{\nu} \), then \( S_1, S_2 \models E[e] \longrightarrow_s S_1, S_3 \mid \bar{\nu} \) with \( I(\bar{\nu}, \emptyset, \bar{\nu}, \emptyset, S_3) \) and \( [S_3]\bar{\nu} = \bar{\nu} \).

**Proof.** By induction on \( [S_1, S_2](E[e]) \longrightarrow^* \bar{\nu} \).

**Case** Reflexive case:
\[
[S_1, S_2](E[e]) = \bar{\nu} \quad \text{(1), given} \\
E = \emptyset, e = \bar{w}, [S_1, S_2]w = \bar{\nu} \quad \text{(2), by (1)} \\
I_E(\bar{w}, \Delta, \Gamma, S_1, S_2) & \quad \text{(3), given} \\
I(\bar{w}, \Delta, \Gamma, S_1, S_2) & \quad \text{(4), by definition} \\
S_1, S_2 | \bar{w} \longrightarrow_s S_1, S_2' | \bar{\nu} & \quad \text{(5), by lemma 8} \\
\bar{\nu} = [S_1, S_2]w = [S_1, S_2']\bar{\nu} & \quad \text{(6), by lemma 8} \\
I(\bar{\nu}, \Delta, \Gamma, S_1, S_2') & \quad \text{(7), by lemma 13} \\
I(\bar{\nu}, \emptyset, \bar{\nu}, \emptyset, S_3) & \quad \text{(8), by lemma 17}
\]

**Case** Transitive case:
\[
[S_1, S_2](E[e]) \longrightarrow e' \quad \text{(1), given} \\
e' \longrightarrow^* \bar{\nu} & \quad \text{(2), given} \\
E'_1[e'_1] = [S_1, S_2](E[e]), e'_1 \longrightarrow e'_2, e' = E'_1[e'_2] & \quad \text{(3), by step} \\
[S_1, S_2]E_1 := E'_1, [S_1, S_2]e_1 := e'_1 & \quad \text{(4), define} \\
E_3[e_1] = E[e] & \quad \text{(5), by (4)} \\
I_E(e, \Delta, \Gamma, S_1, S_2) & \quad \text{(6), given} \\
I_E(e_1, \Delta, \Gamma, S_1, S_2) & \quad \text{(7), by lemma 21} \\
S_1, S_2 | e_1 \longrightarrow^* S_1, S_2' | e'' & \quad \text{(8), by lemma 22} \\
e'_2 := [S_1, S_2]e'' & \quad \text{(9), by lemma 22} \\
I_E(e'', \Delta'', \Gamma, S_1, S_2') & \quad \text{(10), by lemma 23} \\
[S_1, S_2'][E_1[e''']] = ([S_1, S_2']E_1)([S_1, S_2]e'') & \quad \text{(11), commute} \\
= E'_1[e'_2] & \quad \text{(12), by (4) and (9)} \\
e' \longrightarrow^* \bar{\nu} & \quad \text{(13), by (3) and (2)} \\
S_1, S_2 | E_3[e'] \longrightarrow^* S_1, S_3 | \bar{\nu} & \quad \text{(14), inductive hypothesis} \\
I(\bar{\nu}, \emptyset, \bar{\nu}, \emptyset, S_3) = \bar{\nu} & \quad \text{(15), inductive hypothesis} \\
S_1, S_2 | E_1[e_1] \longrightarrow S_1, S_2' | E_1[e''] & \quad \text{(16), eval on (8)} \\
S_1, S_2 | E[e] \longrightarrow^* S_1, S_3 | \bar{\nu} & \quad \text{(17), append (16) and (14)}
\]

**Theorem 7.** (FIP programs are sound in store semantics)
If \( \Delta \models e \) and given disjoint stores \( S_1, S_2 \) with \( \Delta \subseteq \text{dom}(S_1) \), \( S_1 \) sound, \( \Gamma = \text{roots}(S_2) \) and \( S_2 \) linear, and \( [S_1, S_2]e \longrightarrow^* \bar{\nu} \), then \( S_1, S_2 \models e \longrightarrow^* S_1, S_3 \mid \bar{\nu} \) where \( [S_3]\bar{\nu} = \bar{\nu}, \bar{\nu} = \text{roots}(S_3) \) and \( S_3 \) is linear.
Proof.

\[ \Delta \mid \Gamma \vdash e \] (1), given

\( S_1, S_2 \) disjoint stores (2), given

\( \Delta \subseteq \text{dom}(S_1) \) and \( S_1 \) sound (3), given

\( \Gamma = \text{roots}(S_2) \) and \( S_2 \) linear (4), given

\( I(e, \Delta, \Gamma, S_1, S_2) \) (5), by (1)-(4)

\[ e = E[e'] \] (6), for some \( E, e' \)

\( I_E(e', \Delta, \Gamma, S_1, S_2) \) (7), lemma 14

\( S_1, S_2 \mid E[e'] \xrightarrow{s_s^{\ast}} S_1, S_3 \mid \overline{x} \) (8), lemma 24

\( I(\overline{x}, \varnothing, \overline{x}, \varnothing, S_3) \) (9), lemma 24

\[ [S_3]\overline{x} = \overline{v} \] (10), lemma 24

\( \overline{x} = \text{roots}(S_3) \) and \( S_3 \) linear (11), by (9)
We have also re-defined the free variables \( fv \)

**Proof**

If \( \text{Lemma 26.} \)

\( C \) **STACK BOUND**

In this section we wish to prove the stack bound on FIP algorithms.

**C.1 Within a function**

First, we show a bound on the evaluation context when no functions are called by working with a version of the store semantics without the \((\text{anon}_s)\) and \((\text{call}_s)\) rules. In practice, we would not allocate a stack frame for these parts of the evaluation context.

Below, we define the depth \( |e| \) of an expression and the depth \( |E| \) of an evaluation context. We fix a signature \( \Sigma \) and denote by \( |e_{\text{max}}| \) the maximum depth of an expression bound in \( \Sigma \).

\[
\begin{align*}
|x| & := 0 \\
|C^k v_1 \ldots v_k| & := 1 + \max \{ |v_1|, \ldots, |v_k| \} \\
|(v_1, \ldots, v_n)| & := 1 + \max \{ |v_1|, \ldots, |v_n| \} \\
|e_1 e_2| & := 1 + \max \{ |e_1|, |e_2| \} \\
|f(e_1; e_2)| & := 1 + \max \{ |e_1|, |e_2| \} \\
|\text{let } \tilde{x} = e_1 \text{ in } e_2| & := 1 + \max \{ |e_1|, |e_2| \} \\
|\text{match } e \{ p \mapsto \tilde{e} \}| & := 1 + \max \{ |e|, |e_1|, \ldots, |e_n| \} \\
|\text{match! } e \{ p \mapsto \tilde{e} \}| & := 1 + \max \{ |e|, |e_1|, \ldots, |e_n| \} \\
|\text{drop } \tilde{x}; e| & := 1 + |e| \\
|\text{free } k; e| & := 1 + |e| \\
|[\square]| & := 0 \\
|C^k x_1 \ldots E' \ldots v_k| & := 1 + |E'| \\
|(x_1, \ldots, E', \ldots, v_n)| & := 1 + |E'| \\
|E' e| & := 1 + |E'| \\
|x E'| & := 1 + |E'| \\
|f(\tilde{y}; E')| & := 1 + |E'| \\
|\text{let } \tilde{x} = E' \text{ in } e| & := 1 + |E'| \\
|\text{match! } E \{ p \mapsto \tilde{e} \}| & := 1 + |E'| \\
\end{align*}
\]

We also re-define the free variables \( fv \) to also collect all directly called functions

\( fv(f(e_1; e_2)) := \{ f \}, fv(e_1), fv(e_2) \)

**Lemma 25.** *(Relating the depths of expressions and eval contexts)*

We have \( |E[e]| \geq |E| + |e| \).

**Proof.** By straight-forward induction on \( E \).

**Lemma 26.** *(Substitution of variables leaves depth unchanged)*

We have \( |e| = |e[\tilde{x}:=\tilde{y}]| \).

**Proof.** By straight-forward induction on \( e \).

**Lemma 27.** *(Substitution inside eval context)*

If \( |e_1| \geq |e_2| \), then \( |E[e_1]| \geq |E[e_2]| \).

**Proof.** By straight-forward induction on \( E \).

Case \( \square \): By assumption.

Case \( C^k x_1 \ldots E' \ldots v_k \):
\[ |C^k x_1 \ldots E'[e_1] \ldots v_k| = 1 + \max\{|x_1|, \ldots, |E'[e_1]|, \ldots, |v_k|\} \] (1), by definition
\[ |C^k x_1 \ldots E'[e_2] \ldots v_k| = 1 + \max\{|x_1|, \ldots, |E'[e_2]|, \ldots, |v_k|\} \] (2), by definition
\[ |E'[e_1]| \geq |E'[e_2]| \] (3), inductive hypothesis
\[ |E[e_1]| \geq |E[e_2]| \] (4), by (1),(2),(3)

Case \((x_1 \ldots E' \ldots v_k)\):
\[ |(x_1 \ldots E'[e_1] \ldots v_k)| = 1 + \max\{|x_1|, \ldots, |E'[e_1]|, \ldots, |v_k|\} \] (1), by definition
\[ |(x_1 \ldots E'[e_2] \ldots v_k)| = 1 + \max\{|x_1|, \ldots, |E'[e_2]|, \ldots, |v_k|\} \] (2), by definition
\[ |E'[e_1]| \geq |E'[e_2]| \] (3), inductive hypothesis
\[ |E[e_1]| \geq |E[e_2]| \] (4), by (1),(2),(3)

Case \(E' e\):
\[ |E'[e_1]| e| = 1 + \max\{|E'[e_1]|, |e|\} \] (1), by definition
\[ |E'[e_2]| e| = 1 + \max\{|E'[e_2]|, |e|\} \] (2), by definition
\[ |E'[e_1]| \geq |E'[e_2]| \] (3), inductive hypothesis
\[ |E[e_1]| \geq |E[e_2]| \] (4), by (1),(2),(3)

Case \(x E'\):
\[ |x E'[e_1]| = 1 + \max\{|x|, |E'[e_1]|\} \] (1), by definition
\[ |x E'[e_2]| = 1 + \max\{|x|, |E'[e_2]|\} \] (2), by definition
\[ |E'[e_1]| \geq |E'[e_2]| \] (3), inductive hypothesis
\[ |E[e_1]| \geq |E[e_2]| \] (4), by (1),(2),(3)

Case \(f(\overline{y}; E')\):
\[ |f(\overline{y}; E'[e_1])| = 1 + \max\{|\overline{y}|, |E'[e_1]|\} \] (1), by definition
\[ |f(\overline{y}; E'[e_2])| = 1 + \max\{|\overline{y}|, |E'[e_2]|\} \] (2), by definition
\[ |E'[e_1]| \geq |E'[e_2]| \] (3), inductive hypothesis
\[ |E[e_1]| \geq |E[e_2]| \] (4), by (1),(2),(3)

Case let \(\overline{x} = E' e\) :
\[ |\text{let } \overline{x} = E'[e_1] \text{ in } e| = 1 + \max\{|E'[e_1]|, |e|\} \] (1), by definition
\[ |\text{let } \overline{x} = E'[e_2] \text{ in } e| = 1 + \max\{|E'[e_2]|, |e|\} \] (2), by definition
\[ |E'[e_1]| \geq |E'[e_2]| \] (3), inductive hypothesis
\[ |E[e_1]| \geq |E[e_2]| \] (4), by (1),(2),(3)

Case match! \(E \{ p \mapsto e \}\):
\[ |\text{match! } E'[e_1] \{ p \mapsto e \}| = 1 + \max\{|E'[e_1]|, |e_1|, \ldots, |e_n|\} \] (1), by definition
\[ |\text{match! } E'[e_2] \{ p \mapsto e \}| = 1 + \max\{|E'[e_2]|, |e_1|, \ldots, |e_n|\} \] (2), by definition
\[ |E'[e_1]| \geq |E'[e_2]| \] (3), inductive hypothesis
\[ |E[e_1]| \geq |E[e_2]| \] (4), by (1),(2),(3)

Lemma 28. (Store semantics without calls reduces expression depth)
If \(S \mid e \rightarrow_s S' \mid e'\) not using the \((\text{anon}_s), (\text{call}_s)\) rules, then \(|e| \geq |e'|\).

Proof. By induction on the evaluation context \(E\) of the evaluation.

Case \(E = \square\):

Case (let\(_s\)):
\[ |\text{let } \overline{x} = \overline{y} \text{ in } e| = 1 + \max\{|\overline{y}|, |e|\} \] definition
\[ > |e| = |e[\overline{x}:=\overline{y}]| \] using lemma 26

Case (reuse\(_s\)):
At any intermediate step $S$.

Lemma 29.

Case (atom):

$|C| = 1 > 0 = |x|$ definition

Case (bmatch$_i$), (dmatch$_i$):

$|\text{match } y \{ p \rightarrow e \}| = 1 + \max \{ 0, |e_1|, \ldots, |e_n| \}$ definition

$> |e| = |e[\vec{x} := \vec{y}]|$ using lemma 26

Case (dcon$_i$):

$|\text{drop } x; e| = 1 + |e| = |\text{drop } \vec{x}; e|$ definition

Case (free$_i$):

$|\text{free } k; e| = 1 + |e| > |e|$ definition

Case $E \neq \Box$:

$E'[e_1] \geq E'[e_2]$ by inductive hypothesis

$E[e_1] \geq E[e_2]$ using lemma 27

Lemma 29. (Inside a function, the stack bound is $|e|$)

At any intermediate step $S$ | $e[\vec{y} := \vec{y}', \vec{x} := \vec{x}'] \rightarrow S'$ | $E[e']$ not using the (anon$_i$), (call$_i$) rules, we have $|E| + |e'| \leq |e|$.

Proof.

$|e| = |e[\vec{y} := \vec{y}', \vec{x} := \vec{x}']|$ lemma 26

$|e[\vec{y} := \vec{y}', \vec{x} := \vec{x}']| \geq |E[e']|$ repeatly apply lemma 28

$|E[e']| \geq |E| + |e'|$ lemma 25

C.2 First-order

Next, we want to show the usual stack bound in a first order language, omitting just the (anon$_i$) rule this time. As in the paper, we work with a signature $\Sigma$ where all functions $f$ are defined so that their (mutually) recursive calls are in tailposition $T[f]$. However, in a purely first-order sense this tail context is too strict. This is because the side-condition $f_i \notin \text{fv}(e_0)$ also prevents us from storing functions pointers in $e_0$. However, in a first-order language, these function pointers do not matter: We only care about direct calls. We write $\overline{T}[f]$ for the tail context where the side-condition $f_i \notin \text{fv}(e_0)$ is replaced by $f_i \notin \text{calls}(e_0)$ and calls($e_0$) contains all the functions directly called in $e_0$. Clearly, if $e = \overline{T}[f]$, then $e = \overline{T}[\overline{f}]$.

Next, we want to show that the evaluation context does not grow under tail calls. Essentially, this is because the holes in the tail context are precisely the complement of the holes in the evaluation context [Bour et al. 2021].

Lemma 30. (The eval context is complementary to the (first order) tail context)

If $e = \overline{T}[f], f \in \overline{f}$ and $e = E[f(\vec{y}; \vec{x})]$, then $E = \Box$.

If $e = \overline{T}[f], e = E[e']$ and $E \neq \Box$, then $\overline{f} \cap \text{calls}(e') = \emptyset$.

Proof. The eval and tail context are defined as:

$E ::= \Box | \text{match! } E | C^k x_1 \ldots x_k | E \cdot E | f(\vec{y}; E) | \text{let } \vec{x} = E \text{ in } e$

$|e| = 1 + \max \{ |x_1|, \ldots, |x_k| \}$ definition

$C^k x_1 \ldots x_k = 1 > 0 = |x|$ definition

$\boxed{\text{FP}^2: \text{Fully in-Place Functional Programming}}$

$|C^k x_1 \ldots x_k| = 1 + \max \{ |x_1|, \ldots, |x_k| \}$ definition
and
\[
\tilde{T}[\tilde{f}] := e_0 \mid f_i(e_0; e_0) \mid \text{let } \bar{x} = e_0 \text{ in } T[\tilde{f}] \mid \text{match } e_0 \{ p_i \mapsto T_i[\tilde{f}] \} \mid \text{match! } e_0 \{ p_i \mapsto T_i[\tilde{f}] \}
\]
| drop \bar{x}; T[\tilde{f}] \mid \text{free } k; T[\tilde{f}]
\]
\[\text{(where } f_i \not\in \text{ calls}(e_0)\text{)}\]

Unifying \(T[\tilde{f}] = E[f(\bar{y}; \bar{x})]\) by matching on \(T[\tilde{f}]\) yields \(E = \square\) and \(T[\tilde{f}] = f(\bar{y}; \bar{x})\).
Unifying \(T[\tilde{f}] = E[e']\) by matching on \(T[\tilde{f}]\) yields \(f_i \not\in \text{ calls}(e')\) for all \(f_i \in \tilde{f}\).

**Lemma 31.** *(Substitution preserves first order tail contexts)*
If \(e = T[\tilde{f}]\) then \(e[\bar{x}:=\bar{y}] = T[\tilde{f}]\).

**Proof.** By straight-forward induction on \(T[\tilde{f}]\) as the substitution does not act on direct calls.

**Lemma 32.** *(Store semantics preserves first order tail contexts)*
If \(S \mid e \rightarrow_s S' \mid e'\) not using the \((\text{anon}_s), (\text{call}_s)\) rules and \(e = T[\tilde{f}]\) then \(e' = T[\tilde{f}]\).

**Proof.** We use case analysis on \(T[\tilde{f}]\). Let us first assume that \(e = e_0\) where \(f_i \not\in \text{ calls}(e_0)\). By induction on the evaluation context \(E\) of the evaluation.

**Case** \((\text{let}_s)\):
\[
S \mid \text{let } \bar{x} = \bar{y} \text{ in } e \rightarrow_s S \mid e[\bar{x}:=\bar{y}] \quad \text{definition}
\]
\[
\tilde{f} \cap \text{calls}(e) = \emptyset \quad \text{by assumption}
\]
\[
\tilde{f} \cap \text{calls}(e[\bar{x}:=\bar{y}]) = \emptyset \quad \text{by lemma 31}
\]

**Case** \((\text{reuse}_s)\):
\[
S, \circ_k \mid C^k x_1 \ldots x_k \rightarrow_s S, x \mapsto C^k x_1 \ldots x_k \mid x \quad \text{definition, (fresh } x, k \geq 1\)
\]
\[
\text{calls}(x) = \emptyset \quad \text{by definition}
\]

**Case** \((\text{atom}_s)\):
\[
S \mid C \rightarrow_s S, x \mapsto C \mid x \quad \text{definition, (fresh } x\)
\]
\[
\text{calls}(x) = \emptyset \quad \text{by definition}
\]

**Case** \((\text{bmatch}_s)\):
\[
S, y \mapsto C^k \bar{y} \mid \text{match } y \{ p \mapsto e \} \rightarrow_s S, y \mapsto C^k \bar{y} \mid e[y:=\bar{y}] \quad \text{definition, (} p_i = C^k \bar{y}\)
\]
\[
\tilde{f} \cap \text{calls}(e_i) = \emptyset \quad \text{by assumption}
\]
\[
\tilde{f} \cap \text{calls}(e_i[\bar{x}:=\bar{y}]) = \emptyset \quad \text{by lemma 31}
\]

**Case** \((\text{dmatch}_s)\):
\[
S, x \mapsto C^k \bar{y} \mid \text{match! } x \{ p \mapsto e \} \rightarrow_s S, \circ_k \mid e[y:=\bar{y}] \quad \text{definition, (} p_i = C^k \bar{x}\)
\]
\[
\tilde{f} \cap \text{calls}(e_i) = \emptyset \quad \text{by assumption}
\]
\[
\tilde{f} \cap \text{calls}(e_i[\bar{x}:=\bar{y}]) = \emptyset \quad \text{by lemma 31}
\]

**Case** \((\text{dcon}_s)\):
\[
S, x \mapsto C^k \bar{x} \mid \text{drop } x; e \rightarrow_s S \mid \text{drop } \bar{x}; e \quad \text{definition}
\]
\[
\tilde{f} \cap \text{calls}(e) = \emptyset \quad \text{by assumption}
\]
\[
\tilde{f} \cap \text{calls}(\text{drop } \bar{x}; e) = \emptyset \quad \text{by definition}
\]

**Case** \((\text{free}_s)\):
\[
S, \circ_k \mid \text{free } k; e \rightarrow_s S \mid e \quad \text{definition}
\]
\[
\tilde{f} \cap \text{calls}(e) = \emptyset \quad \text{by assumption}
\]

**Case** \((\text{eval})\):
We need to show that $e$ is in a tail-context. We can now assume that $E[e] \not= e_0$. Then split on outermost layer of the evaluation context $E$. First we consider the case where $E \not= \emptyset$.

We can now assume that $E[e] \not= e_0$. Then split on outermost layer of the evaluation context $E$. First we consider the case where $E \not= \emptyset$.

We need to show that $E[e']$ is again a tail-context. We split on the remaining cases of $E$ that match a tail-context $\tilde{T}[\tilde{f}] \not= e_0$:

**Case** $E = f_i(y; \tilde{f})$, $f_i \in f$:

- $f_i(y; e') = \tilde{T}[\tilde{f}]$ (5), given

**Case** $E = \text{let } \bar{x} = \emptyset \text{ in } e_2$:

- let $\bar{x} = e$ in $e_2 = \tilde{T}[\tilde{f}]$ (5), given

**Case** $E = \text{match! } \emptyset \{ p \mapsto e \}$:

- match! $e \{ p \mapsto e \} = \tilde{T}[\tilde{f}]$ (5), given

**Case** $E = \text{let } \bar{x} = \bar{y}$ in $e = \tilde{T}[\tilde{f}]$:

- let $\bar{x} = \bar{y}$ in $e = \tilde{T}[\tilde{f}]$ (5), given

**Case** $E = \text{match! } x \{ p_l \mapsto \tilde{T}[\tilde{f}] \}$:

- match! $x \{ p_l \mapsto \tilde{T}[\tilde{f}] \}$ (5), given

Lastly, assume that $\tilde{T}[\tilde{f}] \not= e_0$ and $E = \emptyset$. Then split on the cases of $\tilde{T}[\tilde{f}]$.

**Case** $f_i(y; \bar{x})$: Impossible, since only the call rule (which we omitted) can reduce in this case.

**Case** $\text{let } \bar{x} = \bar{y}$ in $e = \tilde{T}[\tilde{f}]$:

- let $\bar{x} = \bar{y}$ in $e = \tilde{T}[\tilde{f}]$ (5), given

**Case** $E = \text{match! } x \{ p_l \mapsto \tilde{T}[\tilde{f}] \}$:

- match! $x \{ p_l \mapsto \tilde{T}[\tilde{f}] \}$ (5), given

**Case** $E = \text{drop } x$; $\tilde{T}[\tilde{f}]$:

\[ S \mid E[e] \longrightarrow_s S' \mid E[e'] \] (1), given

\[ \bar{f} \cap \text{calls}(E[e]) = \emptyset \] (2), by assumption

\[ S \mid e \longrightarrow_s S' \mid e' \] (3), (eval) rule

\[ \bar{f} \cap \text{calls}(e') = \emptyset \] (4), inductive hypothesis

\[ \bar{f} \cap \text{calls}(E[e']) = \emptyset \] (5), by (2) and (4)
\[
S, x \mapsto C^k x \mid \text{drop } x; \quad e \xrightarrow{s} S \mid \text{drop } x; \quad e \quad \text{definition}
\]
\[
e = \bar{T}[\bar{f}]
\]
\[
\text{drop } x; \quad e = \bar{T}[\bar{f}] \quad \text{by definition}
\]

**Case** free \(k\); \(\bar{T}[\bar{f}]\):

\[S, \diamond k \mid \text{free } k; \quad e \xrightarrow{s} S \mid e \quad \text{definition}
\]
\[
e = \bar{T}[\bar{f}] \quad \text{assumption}
\]

The store semantics also does not insert calls outside the signature into \(e'\). We show this by showing that it preserves some FIP typing regarding a signature \(\Sigma\).

**Lemma 33.** *(Store semantics preserves FIP typing)*
If \(\Delta \mid \Gamma \vdash \Sigma e\) and \(S \mid e \xrightarrow{s} S' \mid e'\) then \(\Delta' \mid \Gamma' \vdash \Sigma e'\).

**Proof.** By induction on the evaluation context \(E\) in \(e = E[e_1]\). Case \(\square\): By case analysis on the rules of the store semantics.

**Case** \((\text{let}_s)\).

\[
\Delta \mid \Gamma, y \vdash \text{let } x = y \text{ in } e \quad \text{given}
\]
\[
\Delta \mid \Gamma, x \vdash e \quad \text{by LET}
\]
\[
\Delta \mid \Gamma, y \vdash e[x := y] \quad \text{by lemma 11}
\]

**Case** \((\text{call}_s), (\text{anon}_s)\).

\[
\bar{y} \mid \bar{x} \vdash e \quad \text{by DEFFUN and } f(\bar{y}; \bar{x}) = e \in \Sigma
\]
\[
\bar{y}' \mid \bar{x}' \vdash e[\bar{x} := \bar{x}', \bar{y} := \bar{y}'] \quad \text{by lemma 11}
\]

**Case** \((\text{reuse}_s), (\text{atom}_s)\).

\[
\Delta \mid x \vdash x \quad \text{by VAR}
\]

**Case** \((\text{bmatch}_s)\).

\[
\Delta \mid \Gamma \vdash \text{match } y \{ p \mapsto \bar{e} \} \quad \text{given}
\]
\[
\Delta, \text{bv}(p) \mid \Gamma \vdash e \quad \text{by MATCH}
\]
\[
\Delta, \bar{y} \mid \Gamma \vdash e[\text{bv}(p) := \bar{y}] \quad \text{by lemma 11}
\]

**Case** \((\text{dmatch}_n)\).

\[
\Delta \mid \Gamma \vdash \text{match } x \{ p \mapsto \bar{e} \} \quad \text{given}
\]
\[
\Delta \mid \Gamma, \text{bv}(p), \diamond k \vdash e \quad \text{by MATCH!}
\]
\[
\Delta \mid \Gamma, \bar{y}, \diamond k \vdash e[\text{bv}(p) := \bar{y}] \quad \text{by lemma 11}
\]

**Case** \((\text{dcon}_n)\).

\[
\Delta \mid \Gamma, x \vdash \text{drop } x; \quad e \quad \text{given}
\]
\[
\Delta \mid \Gamma, y \vdash \text{drop } \bar{y}; \quad e \quad \text{by DROP}
\]

**Case** \((\text{free}_n)\).

\[
\Delta \mid \Gamma, \diamond k \vdash \text{free } k; \quad e \quad \text{given}
\]
\[
\Delta \mid \Gamma \vdash e \quad \text{by FREE}
\]

**Case** \(C^k x_1 \ldots E' \ldots v_k \mid (x_1, \ldots, E', \ldots, v_n) \mid E' e \mid x E' \mid f(\bar{y}; E') \mid \text{match! } E' \{ p \mapsto \bar{e} \}\):
FP²: Fully in-Place Functional Programming

Case let $\overline{x} = E'$ in $e$: (notice that we may put “children” of $\Gamma_2$ into $\Lambda'$)

$$\Delta \mid \Gamma \vdash E[e]$$ (1), given

$$\Delta, \Gamma_1 \mid \Gamma_1 \vdash E'[e]$$ (2), by (1), where $\Gamma_1, \Gamma_2 \subseteq \Gamma$

$$\Lambda', \Gamma_2 \mid \Gamma_1 \vdash E'[e']$$ (3), inductive hypothesis with $\Lambda' \cap \Gamma_2 = \emptyset$

$$\Delta, \Lambda' \mid \Gamma - \Gamma_1, \Gamma_1 \vdash E'[e']$$ (4), by (3)

We can now prove the main lemma of this section:

**Lemma 34. (First order stack bound)**

Let $\Sigma$ be non-empty and fully-in-place such that for all functions $f \in \Sigma$ mutually recursive with $\overline{f}$ we have $f(\overline{y}; \overline{x}) = \overline{T}[f]$. If $e = \overline{T}[f], \Delta \mid \Sigma e$ and $|e| \leq |e_{\text{max}}|$, then at any intermediate step $S \mid e \rightarrow^* S' \mid E'[e']$ not using the (anon$_t$) rule, we have $|E'| \leq |e_{\text{max}}| \cdot |\Sigma|$.

**Proof**: By induction on $\Sigma$. In both the base and the inductive case, we use another induction on all subderivations of $S \mid e \rightarrow^* S' \mid E'[e']$. Our induction hypothesis is then:

- Either $\Sigma \equiv \overline{f}$ or the lemma holds for all derivations on $\Sigma' \subseteq \Sigma$.
- The lemma holds for all derivations $S_2 \mid e_2 \rightarrow^* S' \mid E'[e']$ such that $S \mid e \rightarrow^* S_2 \mid e_2, e_2 = \overline{T}[f], \Lambda' \mid \Gamma' \vdash \Sigma e_2$ and $|e_2| \leq |e_{\text{max}}|$.

**Case** Base case $S \mid e \rightarrow^* S \mid E'[e']$ with $e = E'[e']$: Clear, since $|E'| \leq |e| \leq |e_{\text{max}}|$ and $|\Sigma| \geq 1$.

If $S \mid e \rightarrow^* S' \mid E'[e']$ even without the call rule, we have

$$|E'| \leq |E'| + |e'| \quad \text{clear}$$

$$\leq |E'[e']| \quad \text{by lemma 25}$$

$$\leq |e| \quad \text{repeatly apply lemma 28}$$

$$\leq |e_{\text{max}}| \quad \text{by assumption}$$

Else, we choose the longest subderivation from $E[e]$ that does not involve the call rule $S \mid e \rightarrow^* S_g \mid E_g[g(\overline{y}_2; \overline{x}_2)]$.

We consider two cases: Either $g$ is in the recursive group $\overline{f}$, or it appears in $\Sigma$ before $\overline{f}$.

**Case** $g$ is in the recursive group, $e = \overline{T}[g]$:

$$E_g[g(\overline{y}_2; \overline{x}_2)] = \overline{T}[g] \quad \text{by lemma 32}$$

$$E_g = \square \quad \text{by lemma 30}$$

$$S_g \mid g(\overline{y}_2; \overline{x}_2) \rightarrow^* S_g \mid e_g[\overline{y}:=\overline{y}_2; \overline{x}:=\overline{x}_2] \quad \text{(call$_g$)}$$

$$e_g = \overline{T}[f] \quad \text{assumption}$$

$$\overline{y} \vdash \overline{x} \vdash e_g \quad \text{since } \Sigma \text{ is fully-in-place}$$

$$|e_g| \leq |e_{\text{max}}| \quad \text{by definition of } e_{\text{max}}$$

$$|E'| \leq |e_{\text{max}}| \cdot |\Sigma| \quad \text{by the inductive hypothesis (2)}$$

**Case** $g$ is not in the same recursive group, $g \in \Sigma - \overline{f}$:

If evaluation never leaves $g$ and $E' = E_g[E''']$, then:

$$|E_g| \leq |e_{\text{max}}| \quad \text{since the derivation involves no (call$_g$)}$$

$$g(\overline{y}_2; \overline{x}_2) = \overline{T}[gs] \quad \text{definition of } \overline{T}[gs], \text{ where } gs \text{ is } g \text{'s recursive group}$$

$$\overline{y}_2 \vdash \overline{x}_2 \vdash g(\overline{y}_2; \overline{x}_2) \quad \text{call}$$

$$|g(\overline{y}_2; \overline{x}_2)| \leq 1 \leq |e_{\text{max}}| \quad \text{definition of length}$$

$$S_g \mid E_g[g(\overline{y}_2; \overline{x}_2)] \rightarrow^* S' \mid E_g[E''[e']] \quad \text{since we chose a subderivation}$$

$$S_g \mid g(\overline{y}_2; \overline{x}_2) \rightarrow^* S' \mid E'[e'] \quad \text{by the (eval) rule}$$

$$|E'| \leq |e_{\text{max}}| \cdot |\Sigma - \overline{f}| \quad \text{by the inductive hypothesis (1)}$$

$$|\Sigma - \overline{f}| + 1 \leq |\Sigma| \quad \text{definition of } |\Sigma - \overline{f}|$$

$$|E'| = |E_g[E''']| = |E_g| + |E'''| \leq |e_{\text{max}}| \cdot |\Sigma| \quad \text{as above}$$

If evaluation leaves $g$ and $S_g \mid E_g[g(\overline{y}_2; \overline{x}_2)] \rightarrow^* S'_g \mid E_g[\overline{x}_3]$, then:
way to access a function pointer

Together, these results imply the well-known stack bound for first-order programs:

**Lemma 35.** *(The first-order stack bound is \(|e_{\text{max}}|\cdot|\Sigma|)*

Let \( \Sigma \) be fully in-place such that for all functions \( f \) in \( \Sigma \) mutually recursive with \( \bar{f} \) we have \( f(\bar{y};\bar{x}) = \bar{T}[\bar{f}] \). Then at any intermediate step \( S \mid f(\bar{y};\bar{x}) \rightarrow_s^* S' \mid E[e'] \) not using the \((\text{anon}_s)\) rule, we have \(|E| \leq |e_{\text{max}}| \cdot |\Sigma|\).

**Proof.** Apply lemma 34 to \( f(\bar{y};\bar{x}) \).

- \( \Sigma \) nonempty since \( f(\bar{y};\bar{x}) = e \in \Sigma \)
- \( f(\bar{y};\bar{x}) = \bar{T}[\bar{f}] \) definition of \( \bar{T}[\bar{f}] \)
- \( \bar{y} \vdash f(\bar{y};\bar{x}) \) \( \text{CALL} \)
- \( |f(\bar{y};\bar{x})| = 1 \leq |e_{\text{max}}| \) definition of length

### C.3 Second-order

Now, we want to get to the main novelty of the proof: Giving a stack bound for anonymous function calls. The proof idea is that our calculus is second-order: An anonymously called function can only make direct calls to other functions (which are mutually recursive, or defined before it). The reason for this is that an anonymous function only receives the store and owned arguments in \( \Gamma \), but we make sure that functions are never in the store or the \( \Gamma \) environment. Then the function has no way to access a function pointer \( f \) defined before itself to call anonymously.

**Lemma 36.** *(The store and \( \Gamma \) never contain functions)*

Let \( \Delta \mid \Gamma \vdash e, g \notin \Gamma, \text{rng}(S) \) and \( S \mid e \rightarrow_s^* S' \mid e' \). Then \( \Delta' \mid \Gamma' \vdash e' \) and \( g \notin \Gamma', \text{rng}(S') \).

**Proof.** By induction on the evaluation context \( E \) in \( e = E[e_1] \). Case \( \Box \): By case analysis on the rules of the store semantics.

**Case (let).**

- \( \Delta \mid \Gamma, \bar{x} \vdash \bar{y} \) in \( e \) given
- \( \Delta \mid \Gamma, \bar{x} \vdash e \) by \( \text{LET} \)
- \( \Delta \mid \Gamma, \bar{y} \vdash e[\bar{x}:=\bar{y}] \) by lemma 11
- \( g \notin \Gamma, \bar{y}, \text{rng}(S) \) by assumption

**Case (call).** \( (\text{anon}_s) \).

- \( \bar{y}' \vdash \bar{x}' \vdash f(\bar{y}';\bar{x}') \) given
- \( \bar{y} \vdash \bar{x} \vdash e \) by \( \text{DEFFUN} \) and \( f(\bar{y};\bar{x}) = e \in \Sigma \)
- \( \bar{y}' \vdash \bar{x}' \vdash e[\bar{x}:=\bar{x}', \bar{y}:=\bar{y}'] \) by lemma 11
- \( g \notin \bar{x}', \text{rng}(S) \) by assumption

**Case (reuse).** \( (\text{atom}_s) \).

- \( \Delta \mid \bar{x} \vdash C^k x_1 \ldots x_k \) by \( \text{REUSE} \)
- \( g \notin \bar{x}, \text{rng}(S) \) given
- \( \Delta \mid x \vdash x \) by \( \text{VAR} \)
- \( g \notin x, \text{rng}(S, x \mapsto C^k x_1 \ldots x_k) \) since \( x \) fresh
We will also need that the tail-context is preserved by evaluation. Here we use the $\text{match}$, which includes the condition, that $g \notin \Gamma$. The stack size would be unbounded. Functions in the same recursive group could call themselves anonymously in non-tail position and not pass along a function pointer of the same recursive group. If we didn't have that restriction, then we could call a tail-call can not pass along a function pointer of the same recursive group. If we didn’t have that restriction, functions in the same recursive group could call themselves anonymously in non-tail position and the stack size would be unbounded.

**Lemma 37.** (*Substitution preserves tail contexts*)

If $e = T[\tilde{f}]$ and $\tilde{f} \cap \overline{y} = \emptyset$, then $e[\overline{\tilde{x}}=\overline{y}] = T[\tilde{f}]$.
Proof. By straight-forward induction on $T[\bar{f}]$, noticing that $\text{fv}(e[\bar{x}:=\bar{y}]) \subseteq \text{fv}(e) \cup \bar{y}$.

Lemma 38. (The eval context is complementary to the tail context)
If $e = T[\bar{f}]$, $\bar{f} \in \bar{f}$ and $e = E[f(\bar{y};\bar{x})]$, then $E = \square$.
If $e = T[\bar{f}]$, $e = E[e']$ and $E \neq \square$, then $\bar{f} \cap \text{fv}(e') = \emptyset$.

Proof. Same as proof of lemma 30.

Lemma 39. (Store semantics preserves tail contexts)
If $S \mid e \longrightarrow_s S' \mid e'$ not using the (anom.) rule, $e = T[\bar{f}]$ and $\bar{f} \cap \text{rng}(S) = \emptyset$, then $e' = T[\bar{f}]$.

Proof. We use case analysis on $T[\bar{f}]$. Let us first assume that $e = e_0$ where $f_i \notin \text{fv}(e_0)$. By induction on the evaluation context $E$ of the evaluation.

Case (let_s):

$S \mid let \ \bar{x} = \bar{y} \in e \longrightarrow_s S \mid e[\bar{x}:=\bar{y}]$ definition

$\bar{f} \cap \bar{y} = \emptyset, \bar{f} \cap \text{fv}(e) = \emptyset$ assumption

$\bar{f} \cap \text{fv}(e[\bar{x}:=\bar{y}]) = \emptyset$ since $\text{fv}(e[\bar{x}:=\bar{y}]) \subseteq \text{fv}(e) \cup \bar{y}$

Case (call_s):

$S \mid f(\bar{y};\bar{x}) \longrightarrow_s S \mid e[\bar{y}:=\bar{y}';\bar{x}:=\bar{x}']$ for $f(\bar{y};\bar{x}) = e \in \Sigma$

$\bar{f} \cap \bar{y}, \bar{x} = \emptyset$ by definition of the tail context

$e = T[\bar{f}]$ by definition of $\Sigma$

$e[\bar{y}:=\bar{y}';\bar{x}:=\bar{x}'] = T[\bar{f}]$ lemma 37

Case (reuse_s):

$S, \circ_k | C^k \ x_1 \ldots x_k \longrightarrow_s S, x \mapsto C^k \ x_1 \ldots x_k | x$ definition, (fresh $x$, $k \geq 1$)

$x \notin \bar{f}$ since $x$ fresh

Case (atom_s):

$S \mid C \longrightarrow_s S, x \mapsto C | x$ definition, (fresh $x$)

$x \notin \bar{f}$ since $x$ fresh

Case (bmatch_s):

$S, y \mapsto C^k \ \bar{y} | match \ y \{p \mapsto e\} \longrightarrow_s S, y \mapsto C^k \ \bar{y} | e_i[\bar{x}:=\bar{y}]$ definition, ($p_i = C^k \bar{y}$)

$\bar{y} \cap \bar{f} = \emptyset$ since $S$ contains no functions

$\bar{f} \cap \text{fv}(e_i) = \emptyset$ by assumption

$\bar{f} \cap \text{fv}(e_i[\bar{x}:=\bar{y}]) = \emptyset$ since $\text{fv}(e[\bar{x}:=\bar{y}]) \subseteq \text{fv}(e_i) \cup \bar{y}$

Case (dmatch_s):

$S, x \mapsto C^k \ \bar{y} | match! \ x \{p \mapsto e\} \longrightarrow_s S, \circ_k | e_i[\bar{x}:=\bar{y}]$ definition, ($p_i = C^k \bar{y}$)

$\bar{y} \cap \bar{f} = \emptyset$ since $S$ contains no functions

$\bar{f} \cap \text{fv}(e_i) = \emptyset$ by assumption

$\bar{f} \cap \text{fv}(e_i[\bar{x}:=\bar{y}]) = \emptyset$ since $\text{fv}(e[\bar{x}:=\bar{y}]) \subseteq \text{fv}(e_i) \cup \bar{y}$

Case (dcon_s):

$S, x \mapsto C^k \ \bar{x} | drop \ x; \ e \longrightarrow_s S | drop \ \bar{x}; \ e$ definition

$\bar{x} \cap \bar{f} = \emptyset$ since $S$ contains no functions

$\bar{f} \cap \text{fv}(e) = \emptyset$ by assumption

$\bar{f} \cap \text{fv}(drop \ \bar{x}; \ e) = \emptyset$ as above
Case (free):
\[ S, \circ_k | \text{free } k; \ e \rightarrow_s S | e \] definition
\[ f \cap \text{fv}(e) = \emptyset \]
by assumption

Case (eval):
\[ S | E[e] \rightarrow_s S' | E[e'] \]
(1), given
\[ f \cap \text{fv}(E[e]) = \emptyset \]
(2), by assumption
\[ S | e \rightarrow_s S' | e' \]
(3), (eval) rule
\[ f \cap \text{fv}(e') = \emptyset \]
(4), inductive hypothesis
\[ f \cap \text{fv}(E[e']) = \emptyset \]
(5), by (2) and (4)

We can now assume that \( T[f] \neq e_0 \). Then split on outermost layer of the evaluation context \( E \).
First we consider the case where \( E \neq \square \).
\[ S | E[e] \rightarrow_s S' | E[e'] \]
(1), given
\[ f \cap \text{fv}(e) = \emptyset \]
(2), by lemma 38
\[ S | e \rightarrow_s S' | e' \]
(3), by (eval)
\[ f \cap \text{fv}(e') = \emptyset \]
(4), by the first case of this proof

We need to show that \( E[e'] \) is again a tail-context. We split on the remaining cases of \( E \) that match a tail-context \( T[f] \neq e_0 \):

Case \( E = g(y; \square) \):
\[ E[e] = T[f] \]
(5), given
\[ f \cap y = \emptyset \]
(6), by (5)
\[ E[e'] = T[f] \]
(7), by (4) and (6)

Case \( E = \text{let } x = y \text{ in } e_2 \):
\[ E[e] = T[f] \]
(5), given
\[ e_2 = T'[f] \]
(6), by (5)
\[ E[e'] = T[f] \]
(7), by (4) and (6)

Case \( E = \text{match } y \{ \text{p}_i \mapsto \text{e}_i \} \):
\[ E[e] = T[f] \]
(5), given
\[ e_i = T_i[f] \]
(6), by (5)
\[ E[e'] = T[f] \]
(7), by (4) and (6)

Lastly, assume that \( T[f] \neq e_0 \) and \( E = \square \). Then split on the cases of \( T[f] \).
Case \( f_i(y; \square) \): Impossible, since only the call rule (which we omitted) can reduce in this case.

Case let \( \overline{x} = \overline{y} \) in \( T[f] \):
\[ S | \text{let } \overline{x} = \overline{y} \text{ in } e \rightarrow_s S | e[\overline{x}:=\overline{y}] \]
definition
\[ \overline{y} \cap f = \emptyset \]
assumption
\[ e = T[f] \]
assumption
\[ e[\overline{x}:=\overline{y}] = T[f] \]
by lemma 37

Case match \( y \{ \text{p}_i \mapsto \text{e}_i \} \):
\[ S, y \mapsto C^k \overline{y} | \text{match } y \{ \text{p} \mapsto \text{e} \} \rightarrow_s S, y \mapsto C^k \overline{y} | e_i[\overline{x}:=\overline{y}] \]
definition, \( p_l = C^k \overline{y} \)
since \( S \) contains no functions
\[ \overline{y} \cap f = \emptyset \]
assumption
\[ e_i = T[f] \]
by lemma 37
Then:

**Proof.** By induction on $S$. In both the base and the inductive case, we use another induction on all subderivations of $S$.

**Case** match! $x \{ p_i \mapsto T_i[\bar{f}] \}$:

- $S, x \mapsto C^k \bar{y}$; match! $x \{ p \mapsto e \} \rightarrow_s S, \circ_k | e_! [\bar{x}:=\bar{y}]$ definition, ($p_i = C^k \bar{x}$)
- $\bar{y} \cap \bar{f} = \emptyset$ since $S$ contains no functions
- $e_i = T_! [\bar{f}]$ assumption
- $e_! [\bar{x}:=\bar{y}] = T_! [\bar{f}]$ by lemma 37

**Case** drop $x$: $T![\bar{f}]$

- $S, x \mapsto C^k \bar{x}$; drop $x$; $e \rightarrow_s S | drop \bar{x}; e$ definition
- $\bar{x} \cap \bar{f} = \emptyset$ since $S$ contains no functions
- $e = T![\bar{f}]$ assumption

**Case** free $k$: $T![\bar{f}]$

- $S, \circ_k | free k; e \rightarrow_s S | e$ definition
- $e = T![\bar{f}]$ assumption

We can now prove the main lemma of this section:

**Lemma 40.** (Second order stack bound)

Let $\Sigma$ be non-empty and fully-in-place such that for all functions $f$ in $\Sigma$ mutually recursive with $\bar{f}$ we have $f(\bar{y}, \bar{x}) = T_! [\bar{f}]$. If $e = T_! [\bar{f}], \Delta | \Gamma \vdash e$ and $|e| \leq |e_{\text{max}}| \cdot |\Sigma|$, and $\Gamma, \text{rng}(S) \cap \text{dom}(\Sigma) = \emptyset$, then at any intermediate step $S | e \rightarrow_s S' | E'[e']$ we have $|E'| \leq |e_{\text{max}}| \cdot |\Sigma|^2$.

**Proof.** By induction on $\Sigma$. In both the base and the inductive case, we use another induction on all subderivations of $S | e \rightarrow_s S' | E'[e']$. Our induction hypothesis is then:

- Either $\Sigma = \bar{f}$ or the lemma holds for all derivations on $\Sigma' \subseteq \Sigma$.
- The lemma holds for $\Sigma$ and all derivations $S_2 | e_2 \rightarrow_s S' | E'[e']$ such that $S | e \rightarrow_s S_2 | e_2$, $e_2 = T_! [\bar{f}], \Delta' | \Gamma' \vdash e_2$, $\text{rng}(S_2) \cap \text{dom}(\Sigma) = \emptyset$, and $|e_2| \leq |e_{\text{max}}| \cdot |\Sigma|$.

**Case** Base case $S | e \rightarrow_s S' | E'[e']$ with $e = E'[e']$: Clear, since $|E'| \leq |e| \leq |e_{\text{max}}| \cdot |\Sigma|$ and $|\Sigma| \geq 1$.

If $S | e \rightarrow_s S' | E'[e']$ even without the anon rule, we have

1. $e = T_! [\bar{f}]$ clear, since $e = T_! [\bar{f}]$
2. $|e| \leq |e_{\text{max}}| \cdot |\Sigma|$ by assumption
3. $|E'| \leq |e_{\text{max}}| \cdot |\Sigma|$ by lemma 34

Else, we choose the longest subderivation from $E[e]$ that does not involve the anon rule $S | e \rightarrow_s S_g | E_g [g(\bar{x}_2)]$. Then:

- $E_g [g(\bar{x}_2)] = T_! [\bar{f}]$ by lemma 39
- $g \notin \bar{f}$ by lemma 38
- $g \in \Sigma^* \rightarrow \bar{f}$ by above

**Case** If evaluation never leaves $g$ and $E' = E_g [E'']$, then:
FP\textsuperscript{2}: Fully in-Place Functional Programming

\[ |E_g| \leq |e_{\max}| \cdot |\Sigma| \]  
\[ g(xs_2) = T[gs] \]  
\[ g \mid xs_2 \vdash g(xs_2) \] since the derivation involves no (\textit{anon}_s)  
\[ x_{s_2}, \text{rng}(S_g) \cap \text{dom}(\Sigma) = \emptyset \]  
\[ |g(xs_2)| = 1 \leq |e_{\max}| \] definition of \( T[gs] \), where \( gs \) is \( g \)'s recursive group \textbf{BAPP}  
\[ E_g[|g(xs_2)|] \rightarrow^* E'[|e'||e'] \] repeatedly apply lemma 36  
\[ S_g \mid E_g[|g(\overline{x}_2)|] \rightarrow^* S' \mid E''[|e''] \] since we chose a subderivation \textbf{by the (eval) rule}  
\[ |E''| \leq |e_{\max}| \cdot |\Sigma - f| \cdot |\Sigma - \overline{f}| \] by the inductive hypothesis (1)  
\[ |\Sigma - \overline{f}| + 1 \leq |\Sigma| \] definition of \( \Sigma - \overline{f} \)  
\[ |E'| = |E_g[E'']| = |E_g| + |E'| \leq |e_{\max}| \cdot |\Sigma| \cdot |\Sigma| \] as above

\textbf{Case} If evaluation leaves \( g \) and \( S_g \mid E_g[|g(\overline{x}_2)|] \rightarrow^* S'_g \mid E_g[\overline{x}_3] \), then:

\[ |E_g[|g(xs_2)|]| \leq |e_{\max}| \cdot |\Sigma| \] since the derivation involves no (\textit{anon}_s)  
\[ |E_g[|xs_3]|| \leq |e_{\max}| \cdot |\Sigma| \] substitution of variables does not increase length  
\[ E_g[|g(xs_2)|] = \mathcal{T}[f] \] by lemma 39  
\[ E_g[|xs_3]| = \mathcal{T}[\overline{f}] \] substitution of variables does not introduce calls  
\[ \Delta' \mid \Gamma' + E_g[|xs_3]| \] repeatedly apply lemma 33  
\[ \Gamma', \text{rng}(S'_g) \cap \text{dom}(\Sigma) = \emptyset \] repeatedly apply lemma 36  
\[ |E'| \leq |e_{\max}| \cdot |\Sigma|^2 \] by the inductive hypothesis (2)

Finally, the theorem:

\textbf{Theorem 8.} \textit{(An FIP program uses constant stack space)}

Let \( \Sigma \) be fully in-place such that for all functions \( f \) in \( \Sigma \) mutually recursive with \( \overline{f} \) we have \( f(y;x) = \mathcal{T}[f] \) and let \( S \) be a store that contains no functions. Then at any intermediate step \( S \mid f(y;x) \rightarrow^* S' \mid E[e'] \), we have \( |E| \leq |e_{\max}| \cdot |\Sigma|^2 \).

\textbf{Proof.} Apply lemma 40 to \( f(y;x) \).

\begin{align*}
\Sigma \text{ nonempty} & \quad \text{since } f(y;x) = e \in \Sigma \\
f(y;x) & = \mathcal{T}[f] \text{ by assumption} \\
\overline{y} & \mid \overline{x} \vdash f(y;x) \text{ \textbf{CALL}} \\
|f(y;x)| & = 1 \leq |e_{\max}| \text{ \textbf{definition of length}} \\
\overline{x}, \text{rng}(S) \cap \text{dom}(\Sigma) & = \emptyset \text{ \textbf{since S contains no functions}}
\end{align*}
\[ f(\overline{y}; \overline{x}) = e \quad \text{and} \quad app(\overline{y}; k, \overline{x'}) = \text{match! } k \]

\[ f'(\overline{y}; \overline{x}, k) = \left[ e \right]_{f,k} \quad \text{and} \quad Acc_E(\Gamma; k') \rightarrow \left[ E_C[\overline{x}] \right]_{f,k'} \]

\[(\text{ctx}) \quad \left[ T[e] \right]_{f,k} = \left[ T \left[ e \right]_{f,k} \right] \]

\[(\text{tail}) \quad \left[ E[f(\overline{y}; e_0)] \right]_{f,k} = f'(\overline{y}; e_0, Acc_E(\Gamma_2; k)) \iff (\star) \]

\[(\text{base}) \quad \left[ e_0 \right]_{f,k} = \text{app}(\overline{y}; k, e_0) \quad \text{where } f \notin \text{fv}(e_0) \]

\[(\star): \text{for } \Delta, \overline{y} \mid \Gamma_1, \Gamma_2 \vdash E[f(\overline{y}; e)], \text{we have } \Delta, \overline{y} \mid \Gamma_1 \vdash f(\overline{y}; e) \text{ and } \overline{y} \mid \Gamma_2 \vdash E[\square] \]

\[ T ::= \square \mid \text{let } \overline{x} = e_0 \text{ in } T \mid \text{match } e_0 \{ p_i \mapsto T_i \} \mid \text{match! } e_0 \{ p_i \mapsto T_i \} \mid \text{drop } \overline{x}; T \mid \text{free } k; T \quad \text{(where } f \notin \text{fv}(e_0)) \]

\[ Acc_E(\varnothing; k') = k' \]

\[ Acc_E(\Gamma, \varnothing, x_1, \ldots, x_n; k') = C_k^E x_1 \ldots x_n Acc_E(\Gamma; k') C_{\text{unit}} \ldots C_{\text{unit}} \text{ where } 0 \leq n < k \]

---

**D TRMC WITH REUSE TRANSLATION**

In this section, we prove the soundness of the TRMC translation with reused context. We base our formalization on the slightly generalised presentation in figure 12. In particular, the constructors of our zipper are generated by an accumulator function \( Acc_E \), which the version in the paper would always call on \( \varnothing, z_i \) with \( k = |z_i| \). But what if \( k \geq |z_i| + 1 \)? We can still represent this by padding the accumulator with atoms \( C_{\text{unit}} \). Similarly, if several reuse credits are available, we create a list of accumulators that jointly stores all free variables.

**D.1 Soundness**

**Lemma 41. (Accumulators are FIP)**

If \( Acc_E(\Gamma; k) \) exists, then \( \varnothing \mid \Gamma, k \vdash Acc_E(\Gamma; k) \).

**Proof.** By induction on \( Acc_E(\Gamma; k) \):

**Case** \( Acc_E(\varnothing; k) = k \): By the \( \text{var} \) rule we have \( \varnothing \mid k \vdash k \).

**Case** \( Acc_E(\Gamma, \varnothing, x_1, \ldots, x_n; k) = C_k^E x_1 \ldots x_n Acc_E(\Gamma; k) C_{\text{unit}} \ldots C_{\text{unit}} \):  
- \( \varnothing \mid \Gamma, k \vdash Acc_E(\Gamma; k) \) \quad (1), by inductive hypothesis
- \( \varnothing \mid \varnothing \vdash C_{\text{unit}} \) \quad (2), \text{ATOM}
- \( \varnothing \mid x_i \vdash x_i \) \quad (3), \text{VAR}
- \( \varnothing \mid \Gamma, \varnothing, x_1, \ldots, x_n, k \vdash C_k^E x_1 \ldots x_n Acc_E(\Gamma; k) C_{\text{unit}} \ldots C_{\text{unit}} \) \quad (4), \text{REUSE}

**Lemma 42. (Matching accumulators is FIP)**

If \( Acc_E(\Gamma; k') \neq k' \) exists and \( \Delta \mid \Gamma_1, \Gamma_2, k' \vdash e \) then \( \Delta \mid \Gamma_1, k \vdash \text{match! } k \{ Acc_E(\Gamma_2; k') \mapsto e \} \).

**Proof.** We recursively expand \( \text{match! } k \{ Acc_E(\Gamma; k') \mapsto e \} \) as:
- \( e[k' := k] \) if \( Acc_E(\varnothing; k') = k' \)
- \( \text{match! } k \{ C_k^E x_1 \ldots x_n k' u_1 \ include drop u_1; \ldots \text{drop } u_m; \text{match! } k' \{ Acc_E(\Gamma; k'') \mapsto e \} \) if \( Acc_E(\Gamma, \varnothing, x_1, \ldots, x_n; k') \text{ exists} \)

Notice that we could also match on the \textit{u} as \textit{match! u \text{C}_{\text{unit}} \mapsto \ldots} instead of dropping them, to obtain a solution without deallocation.
FP²: Fully in-Place Functional Programming

Then, by induction on \( \text{Acc}_E(\Gamma_2; k') \):

Case \( \text{Acc}_E(\emptyset; k') = k' \): By assumption we have \( \Delta | \Gamma_1, \emptyset, k' \vdash e \). Use Lemma 11 to obtain \( \Delta | \Gamma_1, k', k' \vdash e[k' := k] \).

Case \( \text{Acc}_E(\Gamma'_2; k_1, \ldots, x_n; k') = C^k_{\overline{x}} x_1 \ldots x_n \text{Acc}_E(\Gamma'_2; k') \in \text{unit} \ldots \text{unit} \):

\[
\begin{align*}
\Delta & | \Gamma'_1, k' \vdash \text{match}! k' \{ \text{Acc}_E(\Gamma'_2; k') \leftrightarrow e \} \quad (1), \text{by inductive hypothesis} \\
\Delta & | \Gamma'_1, k', u_1, \ldots, u_m \vdash \text{drop} u_1; \ldots \text{drop} u_m; \text{match}! k' \{ \text{Acc}_E(\Gamma; k') \leftrightarrow e \} \quad (2), \text{DROP rule} \\
\Gamma'_1' & = \Gamma_1, \&; k, x_1, \ldots, x_n \quad (3), \text{as } x_i \notin \Gamma_2' \text{ and } \Delta | \Gamma_1', \Gamma_2', k' \vdash e \\
\Delta & | \Gamma_1, k \vdash \text{match}! k \{ C^k_{\overline{x}} x_1 \ldots x_n k' u_1 \ldots u_m \vdash \text{drop} u_1; \ldots \text{drop} u_m; \text{match}! k' \{ \text{Acc}_E(\Gamma; k') \leftrightarrow e \} \quad (4), \text{by MATCH!} \\
\Gamma_2 := \Gamma'_2, \&; k, x_1, \ldots, x_n \quad (5), \text{define} \\
\Delta & | \Gamma_1, k \vdash \text{match}! k \{ \text{Acc}_E(\Gamma_2; k') \leftrightarrow e \} \quad (6), \text{by definition}
\end{align*}
\]

**Lemma 43. (Replacement lemma for tail contexts)**

Let \( \Delta | \Gamma \vdash \overline{T}[e] \) with \( \Delta' | \Gamma' \vdash e \). For all expressions \( e' \) with \( \Delta' | \Gamma', \Gamma'' \vdash e' \), we have \( \Delta | \Gamma, \Gamma'' \vdash \overline{T}[e'] \).

**Proof.** By induction on \( E \).

Case \( \square \): By assumption, as \( \Delta = \Delta' \), \( \Gamma = \Gamma' \).

Case let \( \overline{x} = e_0 \) in \( \overline{T} \):

\[
\begin{align*}
\Delta & | \Gamma_1, \Gamma_2, \Gamma_3 \vdash \overline{T}[e] \quad (1), \text{given} \\
\Delta & | \Gamma_2, \Gamma_3 \vdash \overline{T'}[e] \quad (2), \text{by (1) and LET} \\
\Delta & | \Gamma_2, \Gamma_3, \Gamma'' \vdash \overline{T'}'[e'] \quad (3), \text{by inductive hypothesis} \\
\Delta & | \Gamma_1, \Gamma_2, \Gamma_3, \Gamma'' \vdash \overline{T}[e'] \quad (4), \text{by LET}
\end{align*}
\]

Case match \( e_0 \) \( \{ p \mapsto \overline{T} \} \):

\[
\begin{align*}
\Delta & | \Gamma_1, \Gamma_2 \vdash \overline{T}[e] \quad (1), \text{given} \\
\Delta, \overline{x} & | \Gamma_2 \vdash \overline{T'}[e] \quad (2), \text{by (1) and MATCH!} \\
\Delta & | \Gamma_2, \overline{x}, \Gamma'' \vdash \overline{T'}'[e'] \quad (3), \text{by inductive hypothesis} \\
\Delta & | \Gamma_1, \Gamma_2, \Gamma'' \vdash \overline{T}[e'] \quad (4), \text{by MATCH!}
\end{align*}
\]

Case match! \( e_0 \) \( \{ p \mapsto \overline{T} \} \):

\[
\begin{align*}
\Delta & | \Gamma_1, \Gamma_2 \vdash \overline{T}[e] \quad (1), \text{given} \\
\Delta & | \Gamma_2, \overline{x} \vdash \overline{T'}[e] \quad (2), \text{by (1) and MATCH!} \\
\Delta & | \Gamma_2, \overline{x}, \Gamma'' \vdash \overline{T'}'[e'] \quad (3), \text{by inductive hypothesis} \\
\Delta & | \Gamma_1, \Gamma_2, \Gamma'' \vdash \overline{T}[e'] \quad (4), \text{by MATCH!}
\end{align*}
\]

Case drop \( x; \overline{T} | \text{free } k; \overline{T} \):

\[
\begin{align*}
\Delta & | \Gamma, x \vdash \overline{T}[e] \quad (1), \text{given} \\
\Delta & | \Gamma \vdash \overline{T'}[e] \quad (2), \text{by (1) and the relevant FIP rule} \\
\Delta & | \Gamma, \Gamma'' \vdash \overline{T'}'[e'] \quad (3), \text{by inductive hypothesis} \\
\Delta & | \Gamma, x, \Gamma'' \vdash \overline{T}[e'] \quad (4), \text{by the relevant FIP rule}
\end{align*}
\]

**Lemma 44. (The TRMC translation preserves FIP)**

Fix a function \( f(\overline{y}; \overline{x}) \). If \( \Delta, \overline{y} | \Gamma \vdash e, [e]_{f,k} \) exists and assuming that \( \text{app}(\overline{y}; k, \overline{x}') \) is in the signature, then \( \Delta, \overline{y} | \Gamma, k \vdash [e]_{f,k} \).

**Proof.** Show that the claim holds for any \( \Delta, \Gamma, e \) by induction on the translation rules.

Case \( (tctx) \).
Apply the match!

**Proof** Fix a function \( f \).

**Lemma 45.** (Value replacement lemma)
Let \( \Delta \vdash E[\Box] \) and \( \emptyset \vdash \bar{\nu} \), then \( \Delta \vdash \Gamma, \bar{\nu} \vdash E[\bar{\nu}] \).

**Proof.** By induction on \( E \).

**Case \( \Box \):** By assumption, as \( \Delta, \Gamma = \emptyset \).

**Case** \( C^k x_1 \ldots E' \ldots v_k \mid (x_1, \ldots, E', \ldots, v_n) \mid E' e \mid x E' \mid f(\bar{\nu}; E') \mid \text{match}! E' \{ \bar{p} \mapsto \bar{e} \} \):

\[
\Delta \vdash E[\Box] \quad \text{(1), given}
\]

\[
\Delta \vdash \Gamma, \bar{\nu} \vdash E[\Box] \quad \text{(2), by (1) and the relevant FIP rule}
\]

\[
\Delta \vdash \Gamma, \bar{\nu} \vdash E[\bar{\nu}] \quad \text{(3), by inductive hypothesis}
\]

\[
\Delta \vdash \Gamma, \bar{\nu} \vdash E[\bar{\nu}] \quad \text{(4), by the relevant FIP rule}
\]

**Case** let \( \bar{x} = E' \) in \( e \): (special as we add to the borrowed environment)

\[
\Delta \vdash E[\Box] \quad \text{(1), given}
\]

\[
\Delta, \bar{\nu} \vdash E'[\Box] \quad \text{(2), by (1) and } \text{let}
\]

\[
\Delta, \bar{\nu} \vdash \bar{x} \vdash E'[\bar{x}] \quad \text{(3), by inductive hypothesis}
\]

\[
\Delta \vdash \Gamma, \bar{x} \vdash E[\bar{x}] \quad \text{(4), by } \text{let}
\]

**Lemma 46.** (The \( app \) function is FIP)
Fix a function \( f(\bar{\nu}; \bar{x}) \). If \( \Delta, \bar{\nu} \vdash e \) and \( [e]_{f,k} \) exists, then \( \bar{\nu} \vdash \bar{x}', k \vdash \text{app}(\bar{\nu}; k, \bar{x}') \).

**Proof.** We have:

\[
\text{app}(\bar{\nu}; k, \bar{x}') = \text{match}! k
\]

\[
\text{H} \rightarrow \bar{x}'
\]

\[
\text{Acc}_E(\Gamma; k') \rightarrow [ E_G[\bar{x}'] ]_{f,k'}
\]

Apply the match! rule.

**Case** \( \text{H}: \emptyset \vdash \bar{x}' \vdash \bar{x}' \).

**Case** \( \text{Acc}_E(\Gamma) :\)

\[
\bar{\nu} \vdash E_G[\Box] \quad \text{(1), by } (\ast) \text{ rule}
\]

\[
\bar{\nu} \vdash \Gamma, \bar{x}' \vdash E_G[\bar{x}'] \quad \text{(2), by lemma 45}
\]

\[
\bar{\nu} \vdash \Gamma, \bar{x}', k' \vdash [ E_G[\bar{x}'] ]_{f,k'} \quad \text{(3), by lemma 44}
\]

\[
\bar{\nu} \vdash \bar{x}', k' \vdash \text{match}! k \{ \text{Acc}_E(\Gamma; k') \mapsto [ E_G[\bar{x}'] ]_{f,k'} \} \quad \text{(4), by lemma 42}
\]
FP²: Fully in-Place Functional Programming

**Theorem 9.** (The TRMC transformation is sound.)

Let \( f \) be a function with \( \frac{\text{\( y \mid x \vdash f(y; x) \) and } f(v_1; v_2) \leftarrow^{*} w}{\text{\( y \mid x \vdash f'(y; x, k) \), } y \mid k, x \vdash \text{app}(y; k, x) \text{ and } f'(v_1; v_2, H) \leftarrow^{*} w.} \)

**Proof.** Our transformation follows the general framework of TRMC where we use a defunctionalized evaluation context (section 4.2 of [Leijen and Lorenzen 2023]). Note that we generalize the \((\text{tlet})\) and \((\text{tmatch})\) rules of the TRMC translation (figure 2 in [Leijen and Lorenzen 2023]) to a \((\text{tctx})\) rule. In particular, we include drops and frees in the tail context. However, the correctness of these additional cases is clear. We thus have \( f'(v_1; v_2, H) \leftarrow^{*} w \) by the correctness of the context transformation and the context laws for defunctionalized evaluation contexts.

We have \( y \mid x, k \vdash f'(y; x, k) \) by lemma 44 and \( y \mid k, x' \vdash \text{app}(y; k, x') \) by lemma 46.
E SOUNDNESS OF HEAP SEMANTICS

This section contains the soundness result for the heap semantics. Since the heap semantics extends the store semantics, we will follow the same proof structure. In particular, many lemmata hold unchanged (just replacing \( \rightarrow_s \) by \( \rightarrow_h \), \( S \) with \( H \), using \( \otimes \), etc). Thus we only add the extra cases to each lemma. We extend the free variables as usual:

\[
\begin{align*}
fv(\text{dup} \, \overline{x}; \, e) & \ := \ fv(e), \overline{x} \\
fv(\text{drop} \, \mu \overline{x}; \, e) & \ := \ fv(e), \overline{x} \\
nfv(\text{alloc} \; k; \, e) & \ := \ fv(e) \\
nfv(\lambda \overline{x}. \, e) & \ := \ fv(e) - \overline{x}
\end{align*}
\]

E.1 Properties of Heaps

As before, we will often need to focus on a specific part of the heap that corresponds to all values reachable from a root set \( \Gamma \). In this section, we generalize the properties of stores to heaps using the join operator:

\[
\begin{align*}
\emptyset \otimes H_2 & = H_2 \\
H_1 \otimes_k H_2 & = H_1 \otimes H_2, \otimes_k \\
H_1, x \mapsto^n v \otimes H_2 & = H_1 \otimes H_2, x \mapsto^n v \quad \text{iff} \ x \notin \text{dom}(H_2) \\
H_1, x \mapsto^n v \otimes H_2, x \mapsto^m w, z \mapsto^k w & = H_1 \otimes H_2, x \mapsto^n v, z \mapsto^k w \quad \text{iff} \ \overline{z} = \text{fv}(v)
\end{align*}
\]

This definition still works if \( H_1 \) contains cycles. We can move the elements of the cycle in any order into \( H_2 \). It might be attractive to decrease the counts of \( \overline{z} \) in \( H_1 \) itself, since this would correspond to “deleting” the element \( x \) from \( H_1 \). However, with that modification, we could not handle cycles, because then the children \( \overline{z} \) of the last cycle element would be in \( H_2 \) instead of \( H_1 \).

We write \( \#^x \Gamma \) to denote the number of times \( x \) occurs in \( \Gamma \), \( \#^x H_1 \) to denote the number of times it occurs free in \( \text{rng}(H_1) \) and \( \#^x (H_1 \cap H_2) \) to denote the number of times it occurs free in \( \text{rng}(H_1) \cap \text{rng}(H_2) \).

**Lemma 47.** (Counts in joined heaps)
We have \( \#^x (H_1 \otimes H_2) = \#^x H_1 + \#^x H_2 - \#^x (H_1 \cap H_2) \).

**Proof.** Since the join operator merges compatible \( v \in \text{rng}(H_1) \cap \text{rng}(H_2) \).

**Lemma 48.** (Distributivity of counting)
We have \( \#^x ((H_1 \otimes H_2) \cap H_3) = \#^x (H_1 \cap H_3) + \#^x (H_2 \cap H_3) - \#^x (H_1 \cap H_2 \cap H_3) \).

**Proof.** By lemma 47 on those \( v \) also in \( H_3 \).

**Lemma 49.** (Joining heaps)
Let \( H_1 \) be a sound/linear heap with roots \( \Gamma_1 \) and \( H_2 \) a linear heap with roots \( \Gamma_2 \). Then \( H_1 \otimes H_2 \) is a sound/linear heap with roots \( \Gamma_1, \Gamma_2 \).

**Proof.**

**Case** Well-definedness: The reference counts of the \( \overline{z} \) can become zero temporarily. The count can not become negative, because we only decrease the count of \( \overline{z} \) if their parent is also in \( H_2 \). This can only happen as often as the \( \overline{z} \) occur free in \( \text{rng}(H_2) \). Since \( H_2 \) is linear, the count of \( \overline{z} \) is at least the number of times \( \overline{z} \) occurs free in \( \text{rng}(H_2) \). Furthermore, since \( \overline{z} \) has a reference count of at least one in \( H_1 \), we will end up with a positive reference count in the end.

**Case** Soundness: The invariant that \( x \in \text{dom}(H_1 \otimes H_2) \) iff \( x \in \text{dom}(H_1) \) or \( x \in \text{dom}(H_2) \) is maintained in all cases of the definition \( \otimes \). Since \( H_1 \) and \( H_2 \) are sound, then so is \( H_1 \otimes H_2 \).
We can define a category of linear, compatible heaps, where there is a unique morphism
This shows that the category is monoidal with the join operator as the tensor product and the

\[ n = \#^x \Gamma_1 + \#^x H_1 \]  
\( n + m - \#^x (H_1 \cap H_2) \) (1), since \( H_1 \) linear

\[ m = \#^x \Gamma_2 + \#^x H_2 \]  
\( (2), \text{since } H_2 \text{ linear} \)

\[ n + m - \#^x (H_1 \cap H_2) = \#^x \Gamma_1 + \#^x \Gamma_2 + \#^x H_1 + \#^x H_2 - \#^x (H_1 \cap H_2) \]  
\( (3), \text{final reference count}\)

\[ = \#^x \Gamma_1 + \#^x \Gamma_2 + \#^x (H_1 \otimes H_2) \]  
\( (4), \text{by (1) and (2)} \)

\[ = \#^x \Gamma_1 + \#^x \Gamma_2 + \#^x (H_1 \otimes H_2) \]  
\( (5), \text{by lemma 47} \)

This shows that \( \Gamma_1, \Gamma_2 \) are the roots of \( H_1 \otimes H_2 \).

**Lemma 50. (Associativity of joining heaps)**
Let \( H_1 \) be a sound heap and \( H_2, H_3 \) be linear heaps. Then \((H_1 \otimes H_2) \otimes H_3 = H_1 \otimes (H_2 \otimes H_3)\).

**Proof.** As before \( x \in \text{dom}((H_1 \otimes H_2) \otimes H_3) \) iff \( x \in \text{dom}(H_1), x \in \text{dom}(H_2) \) or \( x \in \text{dom}(H_3) \) iff \( x \in \text{dom}(H_1 \otimes (H_2 \otimes H_3)) \). Let us consider the reference count of one such \( x \) with reference count \( n, m, k \) in \( H_1, H_2, H_3 \) respectively. Then:

\[ n + m + k - \#^x (H_1 \cap H_2) - \#^x ((H_1 \otimes H_2) \cap H_3) \]
\[ = n + m + k - \#^x (H_1 \cap H_2) - \#^x (H_1 \cap H_3) \]
\[ - \#^x (H_2 \cap H_3) + \#^x (H_1 \cap H_2 \cap H_3) \]
\[ = n + m + k - \#^x (H_1 \cap (H_2 \otimes H_3)) - \#^x (H_2 \cap H_3) \]
\( (2), \text{lemma 48} \)

**Lemma 51. (Symmetry of joining heaps)**
Let \( H_1, H_2 \) be linear heaps. Then \( H_1 \otimes H_2 = H_2 \otimes H_1 \).

**Proof.** Clearly, \( \text{dom}(H_1 \otimes H_2) = \text{dom}(H_2), \text{dom}(H_2) = \text{dom}(H_1), \text{dom}(H_1) = \text{dom}(H_2 \otimes H_1) \). By lemma 47, the reference counts are the same.

We can define a category of linear, compatible heaps, where there is a unique morphism \( f : H_1 \rightarrow H_2 \) between \( H_1, H_2 \) iff \( \text{dom}(H_1) \subseteq \text{dom}(H_2) \) and \( \text{roots}(H_1) \subseteq \text{roots}(H_2) \). In that case we just write \( H_1 \rightarrow H_2 \).

**Lemma 52. (Monotocity of joining heaps)**
Let \( H_1 \rightarrow H_2 \) and \( H_2 \rightarrow H_3 \). Then \( H_1 \otimes H_3 \rightarrow H_2 \otimes H_4 \).

**Proof.** As before, the variables of \( H_1 \otimes H_3 \) are those of \( H_1, H_3 \). Those variables occur in \( H_2, H_4 \) and thus \( H_2 \otimes H_4 \). By lemma 49, the roots of \( H_1 \otimes H_3 \) is the disjoint union of the roots of \( H_1 \) and \( H_3 \). By assumption, these are a subset of the roots of \( H_2 \) and \( H_4 \), which give the roots of \( H_2 \otimes H_4 \).

This shows that the category is monoidal with the join operator as the tensor product and the empty heap as tensor unit. For a linear heap with \( \Gamma \subseteq \text{roots}(H) \), we write \( H[\Gamma] \) for the smallest linear subset of \( H \) containing \( \Gamma \). Next we want to define a heap subtraction operator, which will become the internal hom:

\[ [(H_1, \circ_k), (H_2, \circ_k)] = [H_1, H_2] \]
\[ [(H_1, x \mapsto^m v), (H_2, x \mapsto^m v)] = [H_1, H_2] \text{ iff } x \notin \text{fv(rng}(H_2)) \]
\[ [H_1, H_2] = [H_1, H_2] \text{ iff } \text{dom}(H_1) \subseteq \text{fv(rng}(H_2)) \]
\[ [\emptyset, H_2] = H_2 \]
\[ [(H_1, x \mapsto^m v), (H_2, x \mapsto^{m+k} v, z \mapsto^{k+1} w)] = [H_1, (H_2, x \mapsto^m v, z \mapsto^{k+1} w)] \text{ iff } \overline{z} = \text{fv}(v) \]

**Lemma 53. (Heap subtraction)**
If \( H_1 \) and \( H_2 \) are linear heaps with roots \( \Gamma_1, \Gamma_2 \) and \( H_1 \rightarrow H_2 \), then \( [H_1, H_2] \) is a linear heap with roots \( \Gamma_2 - \Gamma_1 \).
Proof.

Case Well-definedness: The operator acts in two stages. In the first stage, all reuse credits are removed from $H_1$. Since roots($H_1$) ⊆ roots($H_2$), each reuse credit in $H_1$ has a matching one in $H_2$. Then we recursively remove all $x \in \text{dom}(H_1)$ which do not occur free in rmg($H_2$). Since $H_1 \rightarrow H_2$, any $x \in \text{dom}(H_1)$ is also in dom($H_2$) with no bigger reference count. In the second stage, we adjust the reference counts of all other such $x$.

Case Soundness: $H_2$ is sound by assumption and we only delete $x$ from $H_2$ if it is not free in rmg($H_2$).

Case Linearity: Clearly, $[H_1, H_2]$ contains all reuse tokens of $H_2$ not contained in $H_1$. Let us focus on the reference counts. After the first stage, $H_1$ and $H_2$ are still linear, even if both have new roots. In the second stage, the reference count of any $x \in \text{dom}(H_2)$ becomes:

$$n = \#^x \Gamma_1 + \#^x H_1 \quad (1), \text{since } H_1 \text{ linear}$$
$$n + m = \#^x \Gamma_2 + \#^x H_2 \quad (2), \text{since } H_2 \text{ linear}$$
$$m + \#^x (H_1 \cap [H_1, H_2]) \quad (3), \text{final reference count}$$
$$= \#^x \Gamma_2 + \#^x H_2 - \#^x \Gamma_1 - \#^x (H_1 \cap [H_1, H_2]) \quad (4), \text{by (1) and (2)}$$
$$= \#^x \Gamma_2 + \#^x H_2 - \#^x \Gamma_1 + \#^x [H_1, H_2] - \#^x (H_1 \otimes [H_1, H_2]) \quad (5), \text{by lemma 47}$$
$$= \#^x \Gamma_2 - \#^x \Gamma_1 + \#^x [H_1, H_2] \quad (6), \text{by lemma 57}$$

Lemma 54. (Hom isomorphism)
If $H_2 \rightarrow H_3$, then $H_1 \otimes H_2 \rightarrow H_3$ iff $H_1 \rightarrow [H_2, H_3]$.

Proof.

Case Domain: We have dom($[H_2, H_3]$) ⊆ dom($H_3$). If dom($H_1$) ⊆ dom($[H_2, H_3]$) and dom($H_2$) ⊆ dom($H_3$), then dom($H_1$), dom($H_2$) ⊆ dom($H_3$).

If $H_1 \otimes H_2 \rightarrow H_3$, then not only dom($H_1$), dom($H_2$) ⊆ dom($H_3$), but in particular any $x \in \text{dom}(H_3)$ will only be deleted in the first phase if not in dom($H_1$): Deletion requires that $x$ and all its parents have the same reference count in $H_2$ and $H_3$, and that $x$ or one of its parents is a root in $H_3$. But then $x$ or one of its parents will also be a root in $H_1$, so the reference count would have to be higher in $H_3$ than in $H_2$. Thus dom($H_1$) ⊆ dom($[H_2, H_3]$).

Case Roots: Let $\Gamma_1, \Gamma_2, \Gamma_3$ be the roots of $H_1, H_2, H_3$ respectively. Clearly, $\Gamma_1 \subseteq \Gamma_3$ iff $\Gamma_1 \subseteq \Gamma_3 - \Gamma_2$.

By lemma 49 the first term gives the roots of $H_1 \otimes H_2$ and by lemma 53 the second term gives the roots of $[H_2, H_3]$.

Lemma 55. (Monotocity of subtracting heaps)
If $H_3 \rightarrow H_1 \rightarrow H_2 \rightarrow H_4$ then $[H_1, H_2] \rightarrow [H_3, H_4]$.

Proof.

Case Domain: The domain of $[H_3, H_4]$ consists of all variables in $H_4$ which are not deleted. Any such variable will be a root of or reachable from a root in $H_3$. Since $H_3 \rightarrow H_1$, the reference counts of $H_1$ are at least as big as those in $H_3$. Thus any deletion in the construction of $[H_3, H_4]$ will also happen during the construction of $[H_1, H_2]$.

Case Roots:

roots($[H_1, H_2]$), roots($H_1$) = roots($H_2$) \hspace{1em} (1), by lemma 53

\subseteq \text{roots($H_4$)} = \text{roots($[H_3, H_4]$)}, \text{roots($H_3$)} \hspace{1em} (2), by lemma 53

\text{roots($H_3$)} \subseteq \text{roots($H_1$)} \hspace{1em} (3), by assumption

\text{roots($[H_1, H_2]$)} \subseteq \text{roots($[H_3, H_4]$)} \hspace{1em} (4), by (2) and (3)

Notice that this does not quite give us a closed monoidal category, since the internal hom $[H_1, H_2]$ is only defined if $H_1 \rightarrow H_2$. Nonetheless, many results about closed monoidal categories also hold in
our setting. As an example of the usefulness of the categorical perspective, we show that subtracting two heaps one-by-one yields the same result as subtracting them as one joined heap:

**Lemma 56.** *(Internalized hom isomorphism)*
If \( H_2 \rightarrow H_3 \) and \( H_1 \rightarrow [H_2, H_3] \), then \( [H_1 \otimes H_2, H_3] = [H_1, [H_2, H_3]] \).

**Proof.** Let \( H \) be a heap with \( H \rightarrow [H_1 \otimes H_2, H_3] \). Then:

\[
\begin{align*}
H \rightarrow [H_1 \otimes H_2, H_3] & \quad \text{(1), by assumption} \\
\text{iff } (H \otimes (H_1 \otimes H_2)) \rightarrow H_3 & \quad \text{(2), by lemma 54} \\
\text{iff } ((H \otimes H_1) \otimes H_2) \rightarrow H_3 & \quad \text{(3), by lemma 50} \\
\text{iff } (H \otimes H_1) \rightarrow [H_2, H_3] & \quad \text{(4), by lemma 54} \\
\text{iff } H \rightarrow [H_1, [H_2, H_3]] & \quad \text{(5), by lemma 54}
\end{align*}
\]

By instantiating \( H \) to \([H_1 \otimes H_2, H_3]\) and \([H_1, [H_2, H_3]]\) using reflexivity, we obtain the equality.

And we can show the existence of the universal morphisms:

**Lemma 57.** *(Existence of counit)*
If \( H_1 \rightarrow H_2 \) then \([H_1, H_2] \otimes H_1 = H_2\).

**Proof.** \([H_1, H_2] \otimes H_1 \rightarrow H_2\): By lemma 54, we have \([H_1, H_2] \otimes H_1 \rightarrow H_2 \) iff \([H_1, H_2] \rightarrow [H_1, H_2] \), which is true by reflexivity.

\( H_2 \rightarrow [H_1, H_2] \otimes H_1 \): We have \( \text{dom}(H) \subseteq (\text{dom}(H_2) \setminus \text{dom}(H_1)), \text{dom}(H_1) \subseteq \text{dom}([H_1, H_2]), \text{dom}(H_1) \).

Additionally, \( \text{roots}(H_2) = (\text{roots}(H_2) - \text{roots}(H_1)), \text{roots}(H_1) \).

**Lemma 58.** *(Existence of unit)*
For any heaps \( H_1, H_2 \) we have \( H_1 \rightarrow [H_2, H_1 \otimes H_2] \).

**Proof.** By lemma 54, we have \( H_1 \rightarrow [H_2, H_1 \otimes H_2] \) iff \( H_1 \otimes H_2 \rightarrow H_1 \otimes H_2 \), which is true by reflexivity.

Notice that the above inequality is strict since \([H_2, H_1 \otimes H_2]\) contains all cycles from \( H_2 \). However, the cycles do not matter too much, since they will also be present in all other heaps which \( H_2 \) was subtracted from:

**Lemma 59.** *(Universal morphism for unit)*
If \( H_2 \rightarrow H_3 \) and \( H_1 \rightarrow [H_2, H_3] \) then \([H_2, H_1 \otimes H_2] \rightarrow [H_2, H_3] \).

**Proof.** We have \( H_1 \otimes H_2 \rightarrow H_3 \) by lemma 54, which implies the claim by lemma 55.

Our last lemma says that if it is possible to subtract \( H_1 \) from \( H_2 \), we can add any \( H_3 \) equally before or after the subtraction:

**Lemma 60.** *(Joining distributes over subtraction)*
If \( H_1 \rightarrow H_2 \) then \([H_1, H_2 \otimes H_3] = [H_1, H_2] \otimes H_3\).

**Proof.**

**Case** Domain: Since \( H_1 \rightarrow H_2 \), an \( x \in \text{dom}(H_3) \) can not get deleted in the construction of \([H_1, H_2 \otimes H_3]\). Thus \( \text{dom}([H_1, H_2 \otimes H_3]) = \text{dom}([H_1, H_2]), \text{dom}(H_3) \).

**Case** Roots:

\[
\begin{align*}
\text{roots}([H_1, H_2 \otimes H_3]) & \quad \text{(1)} \\
= (\text{roots}(H_2), \text{roots}(H_3) - \text{roots}(H_1)) & \quad \text{(2)} \\
= (\text{roots}(H_2) - \text{roots}(H_1)), \text{roots}(H_3) & \quad \text{(3)} \\
= \text{roots}([H_1, H_2] \otimes H_3) & \quad \text{(4)}
\end{align*}
\]
E.2 Simple Invariant

Our simple invariant maintains that the heap remains "well-formed" during evaluation. We use two heaps: a "borrowed" heap $H_1$ which is unchanged by evaluation and an "owned" heap which can be changed. Our simple invariant $I(e, \Delta, \Gamma, H_1, H_2)$ is defined as:

- $\Delta \mid \Gamma \vdash e$
- $H_1, H_2$ are compatible heaps
- $\Delta \subseteq \text{dom}(H_1)$ and $H_1$ sound
- $\Gamma = \text{roots}(H_2)$ and $H_2$ linear

That invariant makes it safe to modify $H_2$, as all values that the heap semantics destroys are in $\Gamma$ and not used anywhere else in the heap ($\Gamma = \text{roots}(H_2)$). It would be enough for soundness to demand $\Gamma \subseteq \text{roots}(H_2)$. However, the stronger assertion directly gives us the garbage-free theorem and by the weakening lemma 64 we can always add separated memory to the heap later.

Unlike in the store semantics, the weakening lemma does not hold for every step. If we have an expression like drop $x$, this will lead to different steps depending on whether the reference count is one or not. Therefore, we need to characterize dropping and dupping in terms of the final heap:

**Lemma 61.** (Dropping joins a root)
If $H \mid \text{dup} x; \ e \overrightarrow{\gamma} H' \mid e$ then $H' = H \otimes H[x]$.

**Proof.** We have $H, x \rightarrow^n v \mid \text{dup} x; \ e \overrightarrow{\gamma} H, x \rightarrow^{n+1} v \mid e$. Since $\text{dom}(H \otimes H[x]) = \text{dom}(H)$ and $\text{roots}(H \otimes H[x]) = \text{roots}(H), x$, we have the claim.

**Lemma 62.** (Dropping subtracts a root)
If $x$ is a root of $H$ and $H' \mid \text{drop} x; \ e \overrightarrow{\gamma} H' \mid e$ then $H' = [H[x], H]$.

**Proof.** We show the statement for all $x$ by induction over all $H$ with $H[x] \rightarrow H$. In the inductive hypothesis, we can assume the claim for all $y$ and $H' \subsetneq H$ with $H'[y] \rightarrow H'$.

**Case** If the reference count of $x$ in $H$ is one, then $H_1, x \overrightarrow{\gamma} v \mid \text{drop} x; \ e \overrightarrow{\gamma} H_1 \mid \text{drop} z; \ e$. Since $H_1 \subsetneq H$ and the $z$ are roots of $H_1$, we can apply the inductive hypothesis to obtain $H' = [H_1[z_1], \ldots [H_1[z_n], H]$.

Since $H[x] = (H_1[z_1] \otimes \ldots \otimes H_1[z_n]), x \overrightarrow{\gamma} v$ this implies the claim.

**Case** If the reference count of $x$ in $H$ is bigger than one, then $H_1, x \overrightarrow{\gamma} v \mid \text{drop} x; \ e \overrightarrow{\gamma} H_1, x \overrightarrow{e} v \mid e$. Since $[H[x], H] = [H[x], H_1], \ldots [H[x], H_n], e$, every variable except $x$ will have an unchanged count and only the count of $x$ will be decreased by one.

**Lemma 63.** (DropRU subtracts a root and adds a credit)
If $H \mid \text{dropRU} x; \ e \overrightarrow{\gamma} H' \mid e$ then $H' = [H[x], H] \otimes o_k$.

**Proof.** The rule for dropRU acts like the rule for drop, except that it puts a space credit into the heap. The claim thus follows from lemma 62.

Then we can weaken an evaluation if it completes all dropping procedures: we over-approximate this by requesting that the final $e'$ does not start with a drop. The interesting part of the proof is that we can move $H_1$ out of the joins and subtractions induced by dups/drops by lemma 50 and 60.

**Lemma 64.** (Weakening for heap semantics)
If $H \mid e \overrightarrow{\gamma} H' \mid e'$ and $e' \notin E[\text{drop} x; \ e'']$, then $H \otimes H_1 \mid e \overrightarrow{\gamma} H' \otimes H_1 \mid e'$ for any compatible $H_1$.

**Proof.** By induction on the judgement $H \mid e \overrightarrow{\gamma} H' \mid e'$. The reflexive case is obvious. Assume that $H \mid e \overrightarrow{\gamma} H' \mid e' \overrightarrow{\gamma} H'' \mid e''$ and $H \otimes H_1 \mid e \overrightarrow{\gamma} H' \otimes H_1 \mid e'$.

Use case analysis on the last step. Then $H' \otimes H_1 \mid e' \overrightarrow{\gamma} H' \otimes H_1 \mid e''$ holds trivially for the cases which do not modify
the heap: \((\text{let}_h), (\text{call}_h), (\text{bmatch}_h), (\text{app}_h)\).

**Case** \((\text{atom}_h, \text{reuse}_h)\).
\[
\begin{align*}
\text{H}' & | E[C^k \ x_1 \ldots x_k] \mapsto_h [\circ_k, \text{H}'], x \mapsto C^k x_1 \ldots x_k | E[x] & (1), \text{by assumption} \\
& = ([\circ_k, \text{H}' \otimes H_i], x \mapsto C^k x_1 \ldots x_k) \otimes H_i & (2), \text{by lemma 60} \\
& = ([\circ_k, \text{H}'], x \mapsto C^k x_1 \ldots x_k) \otimes H_i & (3), \text{since } x \notin \text{dom}(H_i)
\end{align*}
\]

**Case** \((\text{lam}_h)\).
\[
\begin{align*}
\text{H}' & | E[\lambda \bar{x}. e] \mapsto_h [\circ_k, \text{H}'], x \mapsto \lambda \bar{x}. e | E[x] & (1), \text{by assumption} \\
& = ([\circ_k, \text{H}' \otimes H_i], x \mapsto \lambda \bar{x}. e) \otimes H_i & (2), \text{since } x \notin \text{dom}(H_i)
\end{align*}
\]

**Case** \((\text{alloc}_h)\).
\[
\begin{align*}
\text{H}' & | E[\text{alloc } k; e] \mapsto_h \text{H}' \otimes \circ_k | E[e] & (1), \text{by assumption} \\
& = \text{H}' \otimes H_i \circ \circ_k & (2), \text{by lemma 51}
\end{align*}
\]

**Case** \((\text{free}_h)\).
\[
\begin{align*}
\text{H}' & | E[\text{free } k; e] \mapsto_h [\circ_k, \text{H}'] | E[e] & (1), \text{by assumption} \\
& = [\circ_k, \text{H}' \otimes H_i] & (2), \text{by lemma 60}
\end{align*}
\]

**Case** \((\text{dup}_h)\).
\[
\begin{align*}
\text{H}' & | E[\text{dup } x; e] \mapsto_h \text{H}' \otimes \text{H}'[x] | E[e] & (1), \text{by lemma 61} \\
& = \text{H}' \otimes H_i \otimes (\text{H}' \otimes H_i)[x] & (2), \text{since } x \in \text{dom}(H') \\
& = \text{H}' \otimes \text{H}'[x] \otimes H_i & (3), \text{by lemma 51}
\end{align*}
\]

**Case** \((\text{dcon}_h), (\text{dlam}_h), (\text{drop}_h)\).
\[
\begin{align*}
\text{H}' & | E[\text{drop } x; e] \mapsto_h [\text{H}'[x], \text{H}'] | E[e] & (1), \text{by lemma 62} \\
& = [\text{H}'[x], \text{H}' \otimes H_i] & (2), \text{since } x \in \text{dom}(H') \\
& = [\text{H}'[x], \text{H}' \otimes H_i] & (3), \text{by lemma 60}
\end{align*}
\]

**Case** \((\text{dconru}_h), (\text{dropru}_h)\).
\[
\begin{align*}
\text{H}' & | E[\text{dropru } x; e] \mapsto_h [\text{H}'[x], \text{H}' \otimes \circ_k] | E[e] & (1), \text{by lemma 63} \\
& = [\text{H}'[x], \text{H}' \otimes H_i \otimes \circ_k] & (2), \text{since } x \in \text{dom}(H') \\
& = [\text{H}'[x], \text{H}' \otimes H_i \otimes \circ_k] & (3), \text{by lemma 60} \\
& = [\text{H}'[x], \text{H}' \otimes \circ_k \otimes H_i] & (4), \text{by lemma 51}
\end{align*}
\]

### E.3 Progress

In this section we want to show that the heap semantics can progress if the simple invariant is true and operational semantics can progress. We assume throughout that for any function \( f(\overline{y}, \overline{x}) = e \in \Sigma \), we have \( \text{fv}(e) \subseteq \overline{y}, \overline{x} \). This is true if \( \Sigma \) is fully in-place:

**Lemma 65.** (Free variables of FIP expressions are in \( \Delta, \Gamma \))
If \( \Delta | \Gamma \vdash e \), then \( \text{fv}(e) \subseteq \Delta, \Gamma \).

**Proof.** By induction on the judgement \( \Delta | \Gamma \vdash e \).

**Case** \( \text{dup} \):
We define

**Lemma 66.** (Heap Substitution on unused variables)

If \( x \notin \text{fv}(e) \) or \( x \notin \text{dom}(H) \), then \( [H]e = [H - x]e \).

**Proof.** As before

**Lemma 67.** (Heap Substitution commutes)

If \( H \) sound and \( [H]\overline{x} = \overline{v} \), then \( [H](e[\overline{x}:=\overline{z}]) = ([H - \overline{x}]e)[\overline{x}:=\overline{v}] \).

**Proof.** As before

**Lemma 68.** (Heap semantics reads values)

If \( I(\overline{v}, \Delta, \Gamma, H_1, H_2) \) then \( H_1 \otimes H_2 | \overline{v} \rightarrow^* _{\text{h}} H_1 \otimes H_2' \mid \overline{x} \) with \( [H_2]\overline{v} = [H_2']\overline{x} \) and all names in \( \text{dom}(H_2) - \text{dom}(H_2') \).
Proof. Using the \((x_1, \ldots, \square, \ldots, v_0)\) context, view each value \(v\) individually. By induction on \(v\). Unlike in stores, our values now also include lambdas:

\[
\begin{align*}
\text{Case } \lambda \bar{x}. e: \\
\Delta | \bar{x} \vdash \lambda \bar{x}. e & : (1), \text{ by } \text{lam} \\
\bar{x} & = \text{roots}(H) \quad (2), \text{ by invariant} \\
H_1 \otimes H_2 | \lambda \bar{x}. e & \rightarrow_h H_1 \otimes (H_2, x \mapsto \lambda \bar{x}. e) | x \quad (3), \text{ (lam}_h\text{)}, \text{ fresh } x \\
[H_2, x \mapsto \lambda \bar{x}. e]x & = [H_2](\lambda \bar{x}. e) \quad (4), \text{ since } x \text{ fresh}
\end{align*}
\]

Lemma 69. (Dropping can progress)
If \(I((\text{drop } x; e), \Delta, \Gamma, H_1, H_2)\) then \(H_1 \otimes H_2 | \text{ drop } x; e \rightarrow_h^* H_1 \otimes H'_2 | e \text{ with } [H_1 \otimes H_2]e = [H_1 \otimes H'_2]e\).

Proof. Like proof of lemma 70, using lemma 63.

Lemma 71. (Heap semantics can progress (no eval ctxx))
If \(I(e, \Delta, \Gamma, H_1, H_2)\) and \([H_1 \otimes H_2]e \rightarrow e'\), then \(H_1 \otimes H_2 | e \rightarrow_h^* H_1 \otimes H'_2 | e''\) with \(e'' = [H_1 \otimes H'_2]e''\).

Proof. By case-analysis on \([H_1 \otimes H_2]e \rightarrow e'\). We omit the cases which are unchanged from the store semantics.

Case (dup).
\[
H_1 \otimes H_2 | \text{ dup } x; e \rightarrow_h H_1 \otimes H_2 \otimes (H_1 \otimes H_2)[x] | e \quad (1), \text{ by lemma 61} \\
\text{dom}((H_1 \otimes H_2)[x]) \subseteq \text{dom}(H_1 \otimes H_2) \quad (2), \text{ by definition} \\
[H_1 \otimes H_2]e = [H_1 \otimes H_2 \otimes (H_1 \otimes H_2)[x]]e \quad (3), \text{ by (2)}
\]

Case (drop).
\[
H_1 \otimes H_2 | \text{ drop } x; e \rightarrow_h^* H_1 \otimes H'_2 | e \quad (1), \text{ by lemma 69} \\
[H_1 \otimes H'_2]e = [H_1 \otimes H_2]e \quad (2), \text{ by lemma 69}
\]

Case (dropru).
\[
H_1 \otimes H_2 | \text{ dropru } x; e \rightarrow_h^* H_1 \otimes H'_2 | e \quad (1), \text{ by lemma 70} \\
[H_1 \otimes H'_2]e = [H_1 \otimes H_2]e \quad (2), \text{ by lemma 70}
\]

Case (alloc).
\[
H_1 \otimes H_2 | \text{ alloc } k; e \rightarrow_h H_1 \otimes (H_2, \circ k) | e \quad (1), \text{ by } (\text{alloc}_k) \\
[H_1 \otimes H_2]e = [H_1 \otimes (H_2, \circ k)]e \quad (2), \text{ clear}
\]
Case (beta).

\[ H_1 \otimes H_2 \mid (y) \overline{y} \longrightarrow_h H_1 \otimes H_2 \mid \text{dup } \overline{z}; \text{ drop } y; \ e[\overline{x}:=\overline{y}] \]

(1), by \( \text{app}_h \), \( \lambda \overline{x}. \ e \in H_1 \otimes H_2 \)

\[ y, \overline{y} = \text{roots}(H_2) \]

(2), by invariant

\[ (H_1 \otimes H_2)[\overline{z}] = H_2[\overline{z}] \text{ and } (H_1 \otimes H_2)[y] = H_2[y] \]

(3), by (2)

\[ H_1 \otimes H_2 \mid (y) \overline{y} \longrightarrow_h H_1 \otimes [H_2[y], H_2 \otimes H_2[\overline{z}]] \mid e[\overline{x}:=\overline{y}] \]

(4), by (3)

\[ \text{roots}([H_2[y], H_2 \otimes H_2[\overline{z}]]) = \overline{y}, \overline{z} \]

(5), by (2)

\[ \text{fv}(e) = \overline{y}, \overline{z} \]

(6), by lemma 65

\[ e' = [H_1 \otimes H_2']e'' \]

(7), by (5) and (6)

### E.4 Heap semantics preserves linearity and roots

In this section, we wish to maintain the invariant \( I(e, \Delta, \Gamma, H_1, H_2) \) which makes lemma 71 work.

**Lemma 72. (Variable substitution preserves FIP typing)**

Assuming that \( \overline{y} \) does not occur in \( e \). (1) If \( \Delta \mid \Gamma, \overline{x} \vdash e \), then \( \Delta \mid \Gamma, \overline{y} \vdash e[\overline{x}:=\overline{y}] \). (2) If \( \Delta, \overline{x} \mid \Gamma \vdash e \), then \( \Delta, \overline{y} \mid \Gamma \vdash e[\overline{x}:=\overline{y}] \).

**Proof.** By induction on the FIP judgement for any such \( \overline{x}, \overline{y} \).

**Case LAM:**

\[ \Delta \mid \overline{z}, \overline{x} \vdash \lambda \overline{x}. \overline{x}'. \ e \]

(1), given

\[ \emptyset \mid \overline{z}, \overline{x}, \overline{x}' \vdash e \]

(2), by LAM

\[ \overline{z}, \overline{x} = \text{fv}(\lambda \overline{x}'. \ e) \]

(3), by LAM

\[ \emptyset \mid \overline{z}, \overline{y}, \overline{x}' \vdash e[\overline{y}:=\overline{x}] \]

(4), by LAM

\[ \overline{z}, \overline{y} = \text{fv}(\lambda \overline{x}'. \ e[\overline{y}:=\overline{x}]) \]

(5), by (3)

\[ \Delta \mid \overline{z}, \overline{y} \vdash \lambda \overline{x}. \overline{x}'. \ e[\overline{y}:=\overline{x}] \]

(6), by (4),(5)

Other cases are clear.

**Lemma 73. (The delta environment can be weakened)**

If \( \Delta \mid \Gamma \vdash e \), then \( \Delta, x \mid \Gamma \vdash e \).

**Proof.** By straight-forward induction on the judgement \( \Delta \mid \Gamma \vdash e \). Unlike as in the proof of store semantics, we do not have to require \( x \notin \Gamma \) as \( \Delta \) and \( \Gamma \) need not be disjoint.

The next lemma is the main lemma of our soundness proof. It says that the individual steps of the heap semantics take sound/linear heaps to sound/linear heaps. As usual, the judgement \( \longrightarrow_h \) does not include the eval rule. We assume a slightly modified \( \text{app}_h \) rule, where we execute the dup immediately. This is equivalent to the \( \text{app}_h \) rule given in the paper:

\[ (\text{app}_h) \quad H, \overline{z} \mapsto^n \overline{v} \mid (y) \overline{y} \longrightarrow_h H, \overline{z} \mapsto^{n+1} \overline{v} \mid \text{ drop } y; \ e[\overline{x}:=\overline{y}] \quad (y \mapsto^m \lambda \overline{x}. \ e \in H) \]

**Lemma 74. (Heap semantics preserves linearity and roots (no eval ctx))**

If \( I(e, \Delta, \Gamma, H_1, H_2) \) and \( H_1 \otimes H_2 \mid e \longrightarrow_h H_1 \otimes H_3 \mid e', \) then \( I(e', \Delta', \Gamma', H_1, H_3) \).

**Proof.** By case analysis on the rules of the heap semantics.

**Case \( \text{app}_h \).**
FP²: Fully in-Place Functional Programming

\[ \Delta \vdash y, \overline{y} \vdash (y) \overline{y} \]  (1), by `app`
\[ \varnothing \vdash x, \overline{y} \vdash e \]  (2), by `lam`
\[ \varnothing \vdash y, \overline{y} \vdash e[\overline{x}:=\overline{y}] \]  (3), by 72
\[ \varnothing \vdash y, \overline{y}, \overline{z} \vdash \text{drop } y; e[\overline{x}:=\overline{y}] \]  (4), by `drop`
\[ H'_2, \overline{z} \mapsto^y \overline{v} := H_2 \]  (5), define
\[ H_3 := H'_2, \overline{z} \mapsto^{n+1} \overline{v} \]  (6), define
\[ y, \overline{y} = \text{roots}(H_2) \]  (7), given
\[ y, \overline{y}, \overline{z} = \text{roots}(H_3) \]  (8), by (7) and (6)

Case `(alloc_h)`.
\[ \Delta \vdash \Gamma \vdash \text{alloc } k; e \]  (1), given
\[ H_3 := H_2, \circ_k \]  (2), define
\[ \Delta \vdash \Gamma, \circ_k \vdash e \]  (3), by `alloc`
\[ \Gamma = \text{roots}(H_2) \]  (4), by (1)
\[ \Gamma, \circ_k = \text{roots}(H_3) \]  (5), by (2)
\[ H_3 \text{ linear} \]  (6), by (5)

Case `(lam_h)`.
\[ \Delta \vdash \overline{z} \vdash \lambda^x \overline{z}. e \]  (1), given
\[ H_3 := H_2, x \mapsto \lambda^x \overline{z}. e \]  (2), define
\[ \Delta \vdash x \vdash x \]  (3), by `var`
\[ x = \text{roots}(H_3) \]  (4), by (2) and since `x` is fresh
\[ \overline{z} = \text{roots}(H_2) \]  (5), by (1)
\[ H_3 \text{ linear} \]  (6), by (5)

Case `(dcon_h)`.
\[ \Delta \vdash \Gamma, x \vdash \text{drop } x; e \]  (1), given
\[ H_3 := H_2 - x \]  (2), define
\[ x \mapsto^1 C^k \overline{y} \in \text{roots}(H_2) \]  (3), given
\[ H_3 \text{ sound} \]  (4), by (3)
\[ \Gamma, \overline{y} = \text{roots}(H_3) \]  (5), since `H_2` is linear
\[ \Delta \vdash \Gamma, \overline{y} \vdash \text{drop } \overline{y}; e \]  (6), by (1), `drop`

Case `(dconru_h)`.
\[ \Delta \vdash \Gamma, x \vdash \text{dropru } x; e \]  (1), given
\[ H_3 := H_2 - x, \circ_k \]  (2), define
\[ x \mapsto^1 C^k \overline{y} \in \text{roots}(H_2) \]  (3), given
\[ H_3 \text{ sound} \]  (4), by (3)
\[ \Gamma, \circ_k, \overline{y} = \text{roots}(H_3) \]  (5), since `H_2` is linear
\[ \Delta \vdash \Gamma, \overline{y} \vdash \text{drop } \overline{y}; e \]  (6), by (1), `drop`

Case `(dlam_h)`.
\[ \Delta \vdash \Gamma, x \vdash \text{drop } x; e \]  (1), given
\[ H_3 := H_2 - x \]  (2), define
\[ x \mapsto^1 \lambda^x \overline{z}. e \in \text{roots}(H_2) \]  (3), given
\[ H_3 \text{ sound} \]  (4), by (3)
\[ \Gamma, \overline{y} = \text{roots}(H_3) \]  (5), since `H_2` is linear
\[ \Delta \vdash \Gamma, \overline{y} \vdash \text{drop } \overline{z}; e \]  (6), by (1), `drop`

Case `(drop_h)`.
\( \Gamma, x = \text{roots}(H_2) \) (1), given
\( H'_{\delta}, x \mapsto a_{\delta+1}v := H_2 \) (2), define
\( H_3 := H'_\delta, x \mapsto a_{\delta}v \) (3), define
\( \Delta \mid \Gamma \vdash \text{drop } x; e \) (4), given
\( H_3 \) sound (5), by (3)
\( \Gamma = \text{roots}(H_3) \) (6), since \( H_2 \) is linear
\( \Delta \mid \Gamma \vdash e \) (7), by (4)

**Case** (\( \text{dropr}_h \)). Exactly as case (\( \text{drop}_h \)).

**Case** (\( \text{dup}_h \)).
\( \Delta \mid \Gamma \vdash \text{dup } x; e \) (1), given
\( x \in (\Delta, \Gamma) \) (2), by (1)
\( H_3 := H_2 \otimes (H_1 \otimes H_2)[x] \) (3), by lemma 61
\( x \in \text{dom}(H_1 \otimes H_2) \) (4), by (2)
\( H_3 \) linear (5), by (3), (4)
\( \Gamma, x = \text{roots}(H_3) \) (6), where \( \Gamma = \text{roots}(H_2) \)
\( \Delta \mid \Gamma \vdash e \) (7), by (1)

### E.5 Main Invariant

As before, we want to generalize lemma 74 to arbitrary evaluation contexts and use the more complicated invariant. The simple invariant fails here for the same reason it failed in the store semantics – and all results from that section transfer directly to the heap semantics. This is due to the fact that the heap and store semantics use the same evaluation context and borrowing strategy (in the let-bindings).

We define \( I_E(e, \Delta, \Gamma, H_1, H_2) \) by induction on \( E \):

**Case** \( E = \Box \):
\( I_{\Box}(e, \Delta, \Gamma, H_1, H_2) := I(e, \Delta, \Gamma, H_1, H_2) \)

**Case** \( E[E'] = C^E x_1 \ldots E' \ldots v_k \mid (x_1, \ldots, E', \ldots, v_n) \mid E' e \mid x E' \mid f(\overline{y}; E') \mid \text{match } E' \{ \overline{p} \mapsto \bar{e} \} : \)
\( I_{E[E']}(e, \Delta, (\Gamma_i, \Gamma), H_1, H_2) := I_E(e, \Delta', \Gamma_i, (H_1 \otimes [H_2[\Gamma_1], H_2]), H_2[\Gamma_1]) \) and \( I(E[\Box], \Delta, \Gamma, H_1, [H_2[\Gamma_1], H_2]) \)

**Case** \( E[E'] = \) let \( \overline{x} = E' \) in \( e \):
\( I_{E[E']}(e, \Delta, (\Gamma_1, \Gamma_2, \Gamma_3), H_1, H_2) := \)
\( I_{E'[\Box]}(e, \Delta', \Gamma_1, (H_1 \otimes [H_2[\Gamma_1], H_2]), H_2[\Gamma_1]) \)
\( \text{and } I(E[\Box], \Delta, (\Gamma_2, \Gamma_3), H_1, [H_2[\Gamma_1], H_2]) \)

All of lemmas about exchanging the two invariants continue to hold largely unchanged. The only difference is that we have to join \( H_2[\Gamma_1] \) onto \( [H_2[\Gamma_1], H_2] \) using lemma 57, which guarantees that this yields the original \( H_2 \).

**Lemma 75. (Weakening the invariant)**
If \( I(E[e], \Delta, \Gamma, H_1, H_2) \), then \( I_E(e, \Delta, \Gamma, H_1, H_2) \).

**Lemma 76. (Weakening the invariant)**
If \( I_E(E_2[e], \Delta, \Gamma, H_1, H_2) \), then \( I_{E[E_2]}(e, \Delta, \Gamma, H_1, H_2) \).

**Lemma 77. (Values do not borrow)**
If \( \Delta \mid \Gamma \vdash \overline{v} \), then \( \varnothing \mid \Gamma \vdash \overline{v} \).

**Proof.** By induction on \( \Delta \mid \Gamma \vdash \overline{v} \):
**Case** \( \lambda^\overline{\Gamma} \overline{x}. e \):
FP²: Fully in-Place Functional Programming

\[ \Delta \mid \overline{\delta} \vdash \lambda \overline{x}. e \quad (1), \text{by } \text{LAM} \]
\[ \emptyset \mid \overline{x}.e \quad (2), \text{by } (1) \]
\[ \emptyset \mid \overline{x}. \lambda \overline{x}. e \quad (3), \text{by } \text{LAM} \]

**Lemma 78. (Values do not borrow)**
If \( I(\overline{\delta}, \Delta, \Gamma, H_1, H_2) \), then \( I(\overline{\delta}, \emptyset, \Gamma, H'_1, H_2) \) for any \( H'_1 \).

However, notice that we introduce a \( \Delta' \) environment at every level of the evaluation context. That means, that we can only remove a single level of the evaluation context at a time. To model this, we call an evaluation context \( E \) flat if \( E = E'[E'''] \) implies that either \( E' \) or \( E''' \) is the hole. Then:

**Lemma 79. (Strengthening the invariant)**
If \( E \) is flat and \( I_E(\overline{\delta}, \Delta, \Gamma, H_1, H_2) \), then \( I(E[\overline{\delta}], \Delta, \Gamma, H_1, H_2) \).

**Lemma 80. (Strengthening the invariant)**
If \( E_2 \) is flat and \( I_{E_2}(\overline{\delta}, \Delta, \Gamma, H_1, H_2) \), then \( I_E(E_2[\overline{\delta}], \Delta, \Gamma, H_1, H_2) \).

### E.6 Soundness

Now we can show the main soundness theorem. As earlier, we extend the progress and preservation proofs to handle the `step` and `eval` cases. If we evaluate \( e_1 \) under the context \( E_1 \), we need to assume the invariant \( I_{E_1}(e_1, \Delta, \Gamma, H_1, H_2) \). But how can we obtain the invariant for this precise \( E_1 \)? The trick is that we do not have to know \( E_1 \), instead we can just assume \( I_{E_2}(e_2, \Delta, \Gamma, H_1, H_2) \) for \( E_1[e_1] = E_2[e_2] \):

**Lemma 81. (Comparing evaluation contexts)**
Let \( E_1[e_1] = E_2[e_2] \), then either:
- \( E_1 = E'_1[E''] \) with \( E'_1 = E_2 \) and \( e_1 = E''_1[e_1] \)
- \( E_2 = E'_2[E''] \) with \( E'_2 = E_1 \) and \( e_1 = E''_2[e_2] \)

**Lemma 82. (Alignment of invariant)**
If \( e_1 \rightarrow e'_1, E_1[e_1] = E_2[e_2] \) and \( I_{E_1}(e_2, \Delta, \Gamma, H_1, H_2) \), then \( I_{E_1}(e_1, \Delta, \Gamma, H_1, H_2) \).

**Lemma 83. (Heap semantics can progress)**
If \( I_E(e, \Delta, \Gamma, H_1, H_2) \) and \( [H_1 \otimes H_2]e \rightarrow e' \), then \( H_1 \otimes H_2 \mid e \rightarrow \gamma \ H_1 \otimes H'_2 \mid e'' \) with \( e' = [H_1 \otimes H'_2]e'' \).

**Proof.**

\[
I_E(e, \Delta, \Gamma, H_1, H_2)
\]
\[
I(e, \Delta', \Gamma', H'_1, H'_2), \text{ with } H_1 \otimes H_2 = H'_1 \otimes H'_2 \text{ and } H'_2 \subseteq H_2
\]
\[
[H_1 \otimes H_2]e = [H'_1 \otimes H'_2]e
\]
\[
H'_1 \otimes H'_2 \mid e \rightarrow^h \ H'_1 \otimes H'_2 \mid e''
\]
\[
e' = [H'_1 \otimes H'_2]e''
\]
\[
H_3 := H'_3 \otimes [H'_2, H_2]
\]
\[
H_1 \otimes H_2 \mid e \rightarrow^h \ H_1 \otimes H_3 \mid e''
\]
\[
e' = [H_1, H_3]e''
\]

**Lemma 84. (Heap semantics preserves linearity and roots)**
If \( I_E(e, \Delta, \Gamma, H_1, H_2) \) and \( H_1 \otimes H_2 \mid e \rightarrow^h \ H_1 \otimes H_3 \mid e' \), then \( I_E(e', \Delta', \Gamma', H_1, H_3) \).

**Proof.**
Lemma 85. (Soundness lemma)
If $I_E (e, \Delta, \Gamma, H_1, H_2) \text{ and } [H_1 \otimes H_2] (E[e]) \stackrel{\star}{\longrightarrow} \overline{\nu}$, then $H_1 \otimes H_2 \mid e \stackrel{\star}{\longrightarrow}_H H_1 \otimes H_3 \mid \overline{x}$ with $I(\overline{x}, \emptyset, \overline{x}, \emptyset, H_3)$ and $[H_3] \overline{x} = \overline{\nu}$.

Proof. By induction on $[H_1, H_2] (E[e]) \stackrel{\star}{\longrightarrow} \overline{\nu}$.

Case Reflexive case:
- $[H_1 \otimes H_2] (E[e]) = \overline{\nu}$ (1), given
- $E = \Box, e = \overline{\nu}, [H_1 \otimes H_2] \overline{\nu} = \overline{\nu}$ (2), by (1)
- $I_{E1}(\overline{\nu}, \Delta, \Gamma, H_1, H_2)$ (3), given
- $I(\overline{\nu}, \Delta, \Gamma, H_1, H_2)$ (4), by definition
- $H_1 \otimes H_2 \mid \overline{\nu} \stackrel{\star}{\longrightarrow}_{H1} H_1 \otimes H_2 \mid \overline{\nu}$ (5), by lemma 68
- $\overline{\nu} = [H_1 \otimes H_2] \overline{\nu} = [H_1 \otimes H_2] \overline{\nu}$ (6), by lemma 68
- $I(\overline{x}, \emptyset, \overline{x}, \emptyset, H_2)$ (7), by lemma 74
- $I(\overline{x}, \emptyset, \overline{x}, \emptyset, H_2)$ (8), by lemma 78

Case Transitive case:
- $[H_1 \otimes H_2] (E[e]) \longrightarrow e'$ (1), given
- $e' \longrightarrow^* \overline{\nu}$ (2), given
- $E_1[e_1] = [H_1 \otimes H_2] (E[e]), e_1 \longrightarrow e_1', e_1' = E_1[e_1]'$ (3), by step
- $H_1 \otimes H_2 \mid e_1 \stackrel{\star}{\longrightarrow}_{H1} H_1 \otimes H_2 \mid e''$ (8), by lemma 83
- $e_1' = [H_1 \otimes H_2] e''$ (9), by lemma 83
- $I_{E1}(e_1', \Delta', \Gamma', H_1, H_2)$ (10), by lemma 84
- $[H_1 \otimes H_2] (E_1[e_1']) = ([H_1 \otimes H_2] E_1)([H_1 \otimes H_2] e'')$ (11), commute
- $E_1[e_1'] = e_1'$ (12), by (4) and (9)
- $H_1 \otimes H_2 \mid e_1' \stackrel{\star}{\longrightarrow}_{H1} H_1 \otimes H_2 \mid \overline{x}$ (13), by (3) and (2)
- $I(\overline{x}, \emptyset, \overline{x}, \emptyset, H_3)$ (14), inductive hypothesis
- $H_1 \otimes H_2 \mid e_1 \stackrel{\star}{\longrightarrow}_{H1} H_1 \otimes H_2 \mid E_1[e_1']$ (15), inductive hypothesis
- $H_1 \otimes H_2 \mid E[e] \longrightarrow_H H_1 \otimes H_3 \mid \overline{x}$ (16), eval on (8)
- $H_1 \otimes H_2 \mid \overline{x} \longrightarrow_H H_1 \otimes H_3 \mid \overline{x}$ (17), append (16) and (14)

Theorem 10. (FIP programs are sound in heap semantics)
If $\Delta \vdash e$ and given compatible heaps $H_1, H_2$ with $\Delta \subseteq \text{dom}(H_1), H_1$ sound, $\Gamma = \text{roots}(H_2)$ and $H_2$ linear, and $[H_1 \otimes H_2] e \longrightarrow^* \overline{\nu}$, then $H_1 \otimes H_2 \mid e \longrightarrow^*_H H_1 \otimes H_3 \mid \overline{x}$ where $[H_3] \overline{x} = \overline{\nu}, \overline{x} = \text{roots}(H_3)$ and $H_3$ is linear.

Proof.
FP²: Fully in-Place Functional Programming

\begin{align*}
\Delta | \Gamma \vdash e & \quad \text{(1), given} \\
H_1, H_2 \text{ compatible heaps} & \quad \text{(2), given} \\
\Delta \subseteq \dom(H_1) \text{ and } H_1 \text{ sound} & \quad \text{(3), given} \\
\Gamma = \text{ roots}(H_2) \text{ and } H_2 \text{ linear} & \quad \text{(4), given} \\
I(e, \Delta, \Gamma, H_1, H_2) & \quad \text{(5), by (1)-(4)} \\
e = E[e'] & \quad \text{(6), for some } E, e' \\
I_E(e', \Delta, \Gamma, H_1, H_2) & \quad \text{(7), lemma 75} \\
H_1 \otimes H_2 \mid E[e'] \mapsto^*_{h} H_1 \otimes H_3 \mid \overline{x} & \quad \text{(8), lemma 85} \\
I(\overline{x}, \emptyset, \overline{x}, \emptyset, H_3) & \quad \text{(9), lemma 85} \\
[H_3]\overline{x} = \overline{v} & \quad \text{(10), lemma 85} \\
\overline{x} = \text{ roots}(H_3) \text{ and } H_3 \text{ linear} & \quad \text{(11), by (9)}
\end{align*}
Anton Lorenzen, Daan Leijen, and Wouter Swierstra

Received 2023-07-07; accepted 2023-11-07