A Functional Correspondence between Top-down and Bottom-up Tree Algorithms

Fast and Correct Fully In-Place Functions with First-Class Constructor Contexts and Zippers

Microsoft Technical Report, MSR-TR-2023-28, Jul 29, 2023 (v2).

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

Enabled by recent advancements, like fully in-place programming (FIP), and precise reference counting (Perceus), and in combination with *first-class constructor contexts* as presented here, we show how to express various binary search tree algorithms in a purely functional way but with performance that rivals the best C implementations. We illustrate our techniques by studying three particular insertion algorithms on balanced binary seach trees: *move-to-root* trees, *splay* trees, and the recently discovered *zip* trees.

Insertion operations on binary search trees are typically presented in two styles: top-down or bottom-up. In imperative code, these algorithms seem quite different — and, as in the case of splay tree insertion, they may even return slightly different results. However, in a functional setting they are closely related. Using continuation-passing style, defunctionalization and first-class constructor contexts, we show how to transform one into the other, and explain precisely how their difference arises.

We present functional versions, both top-down and bottom-up, of these insertion algorithms, and mechanise the proof of their correspondence using elementary inductive arguments. We build implementations of these algorithms exploiting first-class constructor contexts and fully in-place programming to generate efficient imperative code. Finally, we formalise the published, imperative top-down and bottom-up implementations in the separation logic framework Iris and prove that these are equivalent to the functional versions.

Is there a way to combine the indulgences of impurity with the benefits of purity?
— Phil Wadler [1990]

1 INTRODUCTION

In his book on purely functional datastructures, Okasaki [1999b] showed how we can elegantly express many tree algorithms in a purely functional way. Unfortunately, the performance of such functional algorithms in absolute terms is often worse than their in-place updating imperative counterparts. Enabled by recent advancements, like fully in-place programming (FIP), and precise reference counting (Perceus), and in combination with *first-class constructor contexts* as presented here, we show how to express various binary search tree algorithms in a purely functional way but with performance that rivals the best C implementations. We show that these purely functional implementations give rise to new insights, enable us to prove complex imperative algorithms correct, and even improve upon them.

We illustrate our techniques by studying three particular algorithms on balanced binary seach trees: *move-to-root* trees [Allen and Munro 1978; Stephenson 1980], self-adjusting *splay* trees [Sleator and Tarjan 1985] and the recently discovered randomized *zip* trees [Tarjan et al. 2021]. These imperative tree algorithms are typically presented in two styles: *bottom-up* algorithms traverse the tree in search for a given key and then move back up to the root, performing restructuring on the way up. In contrast, *top-down* algorithms keep a reference to the root and restructure directly as the tree is traversed in search for the key. Upon reaching the key, no further traversal is necessary and the algorithm returns (a pointer to) the tree's root.

In the imperative formulation, these two styles are quite different, but in the functional setting we can see that they are actually closely related: starting from an obvious, yet inefficient specification, we can derive both styles of algorithms using straightforward familiar program transformations, much as Danvy and his collaborators relate abstract machines and evaluators [Ager et al. 2003]. We can formally prove the various definitions are equivalent using elementary inductive arguments. Our key insight is that both styles construct unfinished intermediate trees with a single *hole*. We present two *general techniques* describing how to derive each variant from the simple specification. The goal of each derivation is to ensure each variant is eventually *fully in-place* [Lorenzen et al. 2023b]. Such functions can be executed without needing allocation and using constant stack space, allowing performance that can be competitive with their imperative counterparts (Section 7).

To uncover a bottom-up algorithm, we first apply CPS-conversion to our simple specification and then we defunctionalize the resulting closures [Danvy and Nielsen 2001; Reynolds 1972]. This gives rise to a zipper on binary trees [Huet 1997]. These are known techniques, but we show that the zipper operations can be fully in-place – and we obtain compiled code which mimics the imperative bottom-up algorithms that use pointer-reversal [Schorr and Waite 1967].

But what about deriving top-down algorithms? This seems impossible in a purely functional language, as these algorithms require a reference to the tree's root: rebalancing any subtrees in-place would break referential transparency. We use the *first-class constructor contexts* presented in this paper, to safely encapsulate the required form of mutation. When such contexts are used uniquely, the operations to extend the context or fill its hole are done in constant time. Using *accumulating constructor contexts*, we use equational reasoning to derive an efficient top-down version from the simple recursive specification.

Finally, we can connect our purely functional implementations to their imperative counterparts. To make this precise, we formalize the published, imperative algorithms and prove that they are equivalent to our functional algorithms using the Iris framework for separation logic [Jung et al. 2018]. Our proofs parallel the structure of the functional program and our loop invariant simply describes the arguments of recursive invocations of the functional version. This not only verifies the correctness of the imperative algorithms, but also provides evidence that the techniques presented here indeed elucidate the inductive structure hidden beneath their imperative shell. In particular, we show how various pointer assignments correspond to "hidden" structure that is apparent in the functional description, like constructor contexts, zippers, and tail-calls.

More specifically, the paper makes the following contributions:

- We describe and implement *first-class constructor contexts* which are essential to express efficient top-down algorithms. Rather than relying on a linear type system to ensure constructor contexts are used safely, as proposed by Minamide [1998], we take a different tack. We use a combination of static compilation with runtime support for *context paths* [Leijen and Lorenzen 2023] – constructor contexts now have the benefit of a pure and unrestricted functional interface, whilst still enjoying the indulgence of impure in-place mutation for unique contexts at runtime (Section 3).
- We identify the two general techniques for deriving either a bottom-up or top-down version of a recursive algorithm on binary trees and apply our technique to three different binary search trees: move-to-root trees (Section 2), splay trees (Section 5) and zip trees (Section 6). Deriving bottom-up algorithms uses a defunctionalized CPS transform [Danvy and Nielsen 2001; Reynolds 1972], folklore in the functional programming literature. Such defunctionalized CPS algorithms only tell half the story: we show how we can use accumulating first-class constructor contexts to derive the top-down algorithms. To the best of our knowledge, no bottom-up version of zip trees has been published previously, but we can directly derive such version using our generic techniques.
- Deriving the tree algorithms as fully in-place functional programs uncovers insights that are obscured in their imperative counterparts. For example, we show how move-to-root and splay trees differ on just one essential rebalance step. Also, we show how we can straightforwardly

derive a new imperative top-down zip-tree insertion algorithm from our functional version that improves upon the published one [Tarjan et al. 2021]. Finally, it has been observed that the published top-down and bottom-up insertion algorithms on splay trees are not equivalent [Levy and Tarjan 2019; Lucas 2004]. We precisely describe where this discrepancy arises and give an alternative version of bottom-up splay trees which *is* equivalent to its top-down counterpart.
- We formally prove the correspondence between the top-down and bottom-up algorithms using the Coq proof assistant. First, we prove the functional versions equivalent by a direct proof by induction – which is only possible when working with purely functional specifications. Then we prove the respective versions equivalent to formalizations of the published imperative algorithms using Iris [Jung et al. 2018]. For our proof of the top-down versions, we model constructor contexts using an explicit representation of context paths and show this is expressive enough to model the same sequence of assignments as in the published imperative code.
- We benchmark the performance of each of these derived functional algorithms in the Koka language [Leijen 2021] and compare this to the performance of their counterparts in C, OCaml and Haskell (Section 7). Our fully in-place functional algorithms are as fast as the best implementations in C, even with the additional overhead that Koka incurs due to automatic memory management, header words, arbitrary precision integers, and support for persistence.

2 MOVE-TO-ROOT TREES

Move-to-root trees, independently described by both Stephenson [1980] and Allen and Munro [1978], are a variation of simple binary search trees, where accessing a particular key ensures it moves to the tree's root. By doing so, elements that are accessed often naturally drift to the top of the tree.

2.1 Deriving a Recursive Algorithm

All our examples in this paper are written in the Koka language [Leijen 2014] (v2.4.2) which implements all the features described in this paper (including first-class constructor contexts). We can declare a datatype for binary trees as:

```
type tree
  Node( left : tree, key : key, right : tree )
  Leaf
```

We use an abstract type `key` for the keys stored in the tree but this is usually instantiated to be an `int`. The main operation on binary trees is the `insert` function that takes a tree and a key as its arguments. If the key is not yet in the tree, the `insert` function inserts it; otherwise no elements are inserted or deleted. Crucially, move-to-root trees ensure that the inserted key always ends up at the root of the resulting tree. The resulting tree should still be a binary search tree, hence we can specify the intended behaviour as follows:

```
fun insert( t : tree, k : key ) : tree
  Node( smaller(t,k), k, bigger(t,k) )
```

That is, each call to `insert` should return a new binary search tree storing the elements of `t` smaller than `k`, the key `k`, and the elements of `t` greater than `k`. To complete this specification, we still need to define `smaller` and `bigger`:

```
fun smaller( t : tree, k : key ) : tree
  match t
    Node(l,x,r) ->
      if   x < k then Node(l,x,smaller(r,k))
      elif x > k then smaller(l,k)
      else l
    Leaf -> Leaf
```

```
fun bigger( t : tree, k : key ) : tree
  match t
    Node(l,x,r) ->
      if   x < k then bigger(r,k)
      elif x > k then Node(bigger(l,k),x,r)
      else r
    Leaf -> Leaf
```

This specification captures the essence of move-to-root trees, but it is also quite inefficient, requiring two separate traversals of the input tree. We can obviously do better by fusing these two traversals into a single pass. As a first step, we merge `smaller` and `bigger` into a single function by inlining their definitions into the specification given by the `insert`:

```
fun insert( t : tree, k : key ) : tree
  Node( match t { Node(l,x,r) -> if x < k then Node(l,x,smaller(r,k)) ... }
      , k
      , match t { Node(l,x,r) -> if x < k then bigger(r,k) ...  }
```

Next, we push down the outer `Node` constructor into the branches and `if` statements and merge the common paths together:

```
fun insert( t : tree, k : key ) : tree
  match t
    Node(l,x,r) -> if   x < k then Node( Node(l,x,smaller(r,k)), k, bigger(r,k))
                   elif x > k then Node( smaller(l,k), k, Node(bigger(l,k),x,r))
                   else Node(l,k,r)
    Leaf        -> Node(Leaf,k,Leaf)
```

At this point, the functions still uses `smaller` and `bigger` – we have apparently not made any progress. However, all these calls are on the same subtree in each branch and we simplify this further using our induction hypothesis. Recall our specification that states `fun insert(t,k) = Node( smaller(t,k), k, bigger(t,k))`. We can use this to refine the two calls of the form `Node( Node(l,x,smaller(r,k)), k, bigger(r,k))` into a single recursive call instead:

```
match insert(r,k)
  Node(s,y,b) -> Node( Node(l,x,s), y, b )   // where y == k
```

where we use the variables `s` and `b` for results of calling `smaller` and `bigger`. At this point, we have a complete direct recursive version of `insert`:

```
fun insert( t : tree, k : key )
  match t
    Node(l,x,r) -> if   x < k then match insert(r,k)
                                    Node(s,y,b) -> Node( Node(l,x,s), y, b)
                   elif x > k then match insert(l,k)
                                    Node(s,y,b) -> Node( s, y, Node(b,x,r))
                   else Node(l,k,r)
    Leaf        -> Node(Leaf,k,Leaf)
```

This version performs a single traversal over the input tree. As a pure functional implementation, it is straightforward to verify the correctness properties of this function in proof assistants such as Coq [2017] – we discuss this in detail in Section 4. Some properties that we can prove this way include:

- whenever `t` is a binary search tree, so is `insert(t,k)`;
- every key in `t` also occurs in `insert(t,k)`; and
- the key stored at the root of `insert(t,k)` is equal to `k`.

However, the current definition of `insert` is not tail-recursive and can use stack space linear in the size of the tree in the worst-case. We will now proceed to derive efficient *fully in-place* bottom-up and top-down variants that remedy these issues.

2.2 Bottom-Up Move-To-Root

A bottom-up algorithm first traverses down the tree to the insertion point, and then restructures the tree on the way back up. We derive a bottom-up traversal from our derived recursive function using standard techniques: a CPS transformation [Ager et al. 2003; Danvy et al. 2007; Plotkin 1975] followed by defunctionalization [Bell et al. 1997; Danvy and Nielsen 2001]of the closures. Less standard however, is our use of *fully in-place functional programming* [Lorenzen et al. 2023b] to ensure that the resulting algorithm can be executed using destructive updates when possible.

CPS Conversion. In our derived recursive version we match on the recursive call as:

```
... if x < k then match insert(r,k) { Node(s,y,b) -> Node(Node(l,x,s),y,b) }
```

We can apply a CPS conversion to make it tail-recursive – instead of matching on the result, we pass a *continuation* `cont` instead:

```
fun insert-cps( t : tree, k : key )
  down-cps(t,k,id)
```

```
fun down-cps( t : tree, k : key, cont : tree -> exn tree ) : exn tree
  match t
    Node(l,x,r) -> if   x < k then down-cps( r, k, fn(t) match t
                                                  Node(s,y,b) -> cont(Node(Node(l,x,s),y,b)))

                   elif x > k then down-cps( l, k, fn(t) match t
                                                  Node(s,y,b) -> cont(Node(s,y,Node(b,x,r))))
                   else cont(Node(l,k,r))
    Leaf        -> cont(Node(Leaf,k,Leaf))
```

Defunctionalized CPS. The `down-cps` function uses three higher order continuation arguments: one that traverses the right subtree; one that traverses the left subtree; and the identity function used as the initial continuation. We now *defunctionalize* these, turning functions into constructors storing the free variables of each continuation [Reynolds 1972]:

```
type zipper
  Done
  NodeL(up : zipper, key : key, right : tree )
  NodeR(left : tree, key : key, up : zipper )
```

We have purposefully named the defunctionalized data type `zipper`, as it corresponds to the *zipper* or 'one hole context' on binary trees [Huet 1997; McBride 2001]. Although we can no longer apply the continuations directly, we can dispatch on the constructors to construct the desired tree:

```
fip fun rebuild( z : zipper, t : tree )
  match z
    Done -> t
    NodeR(l,x,up) -> match t                              // we came from the right
      Node(s,y,b) -> rebuild( up, Node( Node(l,x,s), y, b))
    NodeL(up,x,r) -> match t                              // we came from the left
      Node(s,y,b) -> rebuild( up, Node( s, y, Node(b,x,r)))
```

The `rebuild` function repeatedly moves up through the zipper reassembling the original tree. This function is *fully in-place* as indicated by the `fip` keyword – a point we return to in the next section. Using these definitions, we now derive the complete tail-recursive bottom-up insert function:

```
fip(1) fun insert-bu( t : tree, k : key )
  down-bu(t,k,Done)

fip(1) fun down-bu( t : tree, k : key, z : zipper )
  match t
    Node(l,x,r) -> if   x < k then down-bu( r, k, NodeR(l,x,z) )
                   elif x > k then down-bu( l, k, NodeL(z,x,r) )
                   else rebuild( z, Node(l,k,r) )
    Leaf        -> rebuild( z, Node(Leaf,k,Leaf) )
```

The `insert-bu` function is both *correct* and *fast*. The `fip(1)` annotation allows allocation of a single constructor when the inserted key is not in the tree yet, but no other memory needs to be allocated. It is also correct as we derived it equationally from the recursive specification. We can also prove this more formally:

Theorem 1. (*Correctness of bottom-up move-to-root insert*)
The recursive version of move-to-root trees calculated at the beginning of this section and the bottom up version coincide:

```
down-bu(t,k,z)  ≡  rebuild(z, insert(t,k))
```

Proof. The proof proceeds by induction on the tree `t`. The base case, when `t` is a leaf, is trivial. If the tree is non-empty, we distinguish three cases, depending on the key `x` stored at `t` is less than, greater than, or equal to `k`. We cover the first case – the others are similar – where we need to show:

```
down-bu(r, k, NodeR(l,x,z))  ≡  rebuild(insert(r,k), NodeR(l,x,z))
```

which follows immediately from our induction hypothesis.
An obvious corollary of this theorem is that the recursive version calculated at the beginning of this section coincides with the tail-recursive bottom-up insert function presented here:

Corollary 1. (*Bottom-up is correct*)
For all trees `t` and keys `k`, we have `insert-bu(t,k) ≡ insert(t,k)`.

2.3 Intermezzo: Fully in Place Functional Programming

Throughout this paper, we write purely functional programs, but the goal is always to derive *fully in-place*, or `fip`, functions that can be compiled to efficient code. This section highlights the key principles underlying this recent paradigm of fully-in place functional programming. Consider for example the function that swaps the left and right subtrees:

```
fip fun swap (t : tree) : tree
  match t
    Leaf        -> Leaf
    Node(l,x,r) -> Node(r,x,l)
```

In recent work, Lorenzen et al. [2023b] define a simple linear calculus. Any program in that fragment can be compiled to code that does not use any (de)allocation and uses constant stack space: it can be executed fully in-place. The `fip` keyword asserts that a given function is in this linear fragment, where the Koka compiler statically checks that each `fip` function does not duplicate or discard its arguments; when a function is erroneously marked as `fip`, the Koka compiler gives a warning statically. In the swap function, for instance, we can *reuse* the heap cell from the pattern match to "allocate" the new `Node` constructor. The compiled code would look like:

```
fip fun swap(t : tree) : tree
  match t
    Leaf        -> Leaf
    Node(l,x,r) -> val p = &t in Node@p(r,x,l)
```

Here `&t` takes the heap address of `t`, and the annotation `Node@p` initializes a `Node` cell at address `p`. The final code therefore swaps the pointers in the `Node` cell in-place – just as a C programmer might write. Intuitively, we can check if a function is fip by ensuring that in each branch the constructors matched on use the same memory layout as the constructors "allocated", ensuring every heap cell can be reused. The reuse analysis allows constructors from different datatypes to reuse the same memory cells, illustrated by the following case from the earlier `down-bu` function:

```
Node(l,x,r) -> if   x < k then down-bu( r, k, NodeR(l,x,z))
```

Formally, in each branch of a case expression, the constructor that is matched provides us with a *reuse credit* of a certain size k, written as $\diamond_k$ (similar to the space credits of Aspinall et al. [2008]). These reuse credits are discharged when space of that size is required: in `down-bu` the `NodeR(l,x,z)` discharges the reuse credit $\diamond_3$ obtained by matching on the `Node` constructor. Constructors without arguments, like `Nil`, `True`, or `Leaf`, and primitive types like integers, are called *atoms* and require no allocation. Furthermore, value types like tuples are always unboxed and passed as registers or on the stack.

Nevertheless, it is only safe to reuse these memory locations if the original parameters are *owned* and unique at runtime! Inside `fip` functions the linear use of owned parameters is guaranteed, but when fip functions are called from a non-fip context, the arguments may be shared. Consider the following example:

```
fun mirror( t : tree, k : key ) : tree
  Node(t,k,swap(t))
```

Here the tree `t` is now shared. Any in-place update on `t` would be unsound and change the meaning of this program. To ensure `fip` functions are executed safely, Koka uses precise reference counting [Reinking, Xie et al. 2021; Ullrich and de Moura 2019] to determine dynamically whether or not arguments can be reused in-place. In particular, for a function like `swap`, the generated code becomes:

```
fip fun swap(t : tree) : tree
  match t
    Leaf        -> Leaf
    Node(l,x,r) -> val p = if unique(t) then &t else {dup(l); dup(r); decref(t); alloc(3)}
                   in Node@p(r,x,l)
```

That is, if `t` does have a unique reference count of 1 at runtime, the allocated space is reused. Otherwise, `t` is shared: the reference counts are adjusted and a fresh heap cell is allocated.

The `fip` annotation in Koka only guarantees that no (de)allocation occurs if the parameters are unique at runtime. This may be viewed as weakness – we do not guarantee statically that a function will actually be executed in-place – but it does offer greater flexibility where we can use `fip` functions in both modes. In particular, for the tree algorithms in this paper, we not only get the efficient in-place updating behaviour for unique trees, but we can also use them *persistently* where any shared (sub)trees are copied as needed. Moreover, as we see in the benchmarks in Section 7, the overhead of the dynamic uniqueness test is small and performance is close to that of the corresponding in-place updating C algorithms (which do not support persistent usage).

The `fip` check provides a strong guarantee: constant stack usage and no (de)allocation at all. Throughout this paper, we also use the `fip(n)` variant which allows a function to allocate at most `n` constructors. This is useful for tree insertion algorithms, as we may need to allocate a constant amount of memory for the single node storing the new key. We also have the weaker `fbip` annotation for functions that do not allocate, but are allowed to deallocate and recurse on the stack.

2.3.1 Improving Bottom-Up. The `swap` function may seem trivial – but consider the following slight variation that rotates a binary tree, moving subtrees from the left to the right:

```
fip fun rotate-right( t : tree ) : tree
  match t
    Node(Node(ll,lx,lr),x,r) -> Node(ll,lx,Node(lr,x,r))
    Node(Leaf,x,Leaf)        -> Node(Leaf,x,Leaf)
    Leaf                     -> Leaf
```

It is easy to check that this function is fully in-place. As `fip` functions can safely call other `fip` functions, we can rewrite our `rebuild` function as follows:

```
fip fun rebuild( z : zipper, t : tree ) : tree
  match z
    Done -> t
    NodeL(up,x,r) -> rebuild( up, rotate-right(Node(t,x,r)) )
    NodeR(l,x,up) -> rebuild( up, rotate-left(Node(l,x,t)) )
```

This now corresponds closely to the published algorithm by Allen and Munro [1978] where they also use a bottom-up traversal using left- and right rotations. We formalise the precise relation between our bottom-up `insert-bu` function and the published algorithms shortly (Section 4), but first turn our attention to the top-down version of the same algorithm.

3 FIRST-CLASS CONSTRUCTOR CONTEXTS

A top-down algorithm traverses a structure down in a single pass and directly returns the result structure when reaching the final position. Unfortunately, in a purely functional language is not quite possible to express such algorithms directly. Consider the `map` function for example[1]:

```
fip fun map( xs : list<a>, ^f : a -> b ) : list<b>
  match xs
    Cons(x,xx) -> Cons( f(x), map(xx,f) )
    Nil        -> Nil
```

Naively, this function would use stack space linear in the size of the first list. A well-known solution to derive a tail-recursive version is to use an *accumulator* for the result list, as:

[1]The notation `^f` borrows the `f` parameter so we can use it non-linearly inside the `fip` function.

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

fip fun map(xs,f)
  map-acc(xs,f,Nil)
```

This is not quite a top-down version, though, as eventually we need to traverse back through the accumulator to reverse it (in $O(n)$ time). Similarly, using a function (or difference list [Clark and Tärnlund 1977; Hughes 1986]) as the accumulator, we eventually need to apply the function which essentially traverses back up through the compositions to reconstruct the final list.

In order to be able to express true top-down algorithms, we introduce *first-class constructor contexts*. This abstraction can safely encapsulate the limited form of mutation necessary to define top-down algorithms, while still having a purely functional interface. We define a *constructor context* as a sequence of constructor applications that ends in a single *hole*. We can describe such contexts using the following grammar:

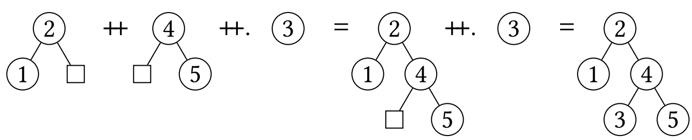
$$v \ ::= \ldots \mid \mathsf{ctx}\ \mathsf{K} \qquad\qquad \mathsf{K} \ ::= \square \mid C^k\ v_1 \ldots \mathsf{K} \ldots v_k$$

where we use v for values, and C^k for a constructor that takes k arguments. In Koka, the keyword `ctx` starts a constructor context and the hole $\square$ is written as an underscore `_`. For example, we can write a list ending in a hole as `ctx Cons(1,_)`, or a binary tree with a hole as `ctx Node(Node(Leaf,1,Leaf),2,_)`.

Constructor contexts support two operations: we can compose (or "append") two contexts as $c_1 \mathbin{+\!\!+} c_2$, and apply a value to a context to fill the hole as $c \mathbin{+\!\!+.}\ v$. Although they can also be implemented as a regular datatype, we shall treat these constructor contexts as an *abstract type*. For example, `(ctx Cons(1,_)) ++ (ctx Cons(2,_)) ++. Nil` evaluates to `(ctx Cons(1,Cons(2,_))) ++. Nil` and then to `[1,2]`. Similarly, the expression:

```
(ctx Node(Node(Leaf,1,Leaf),2,_)) ++ (ctx Node(_,4,Node(Leaf,5,Leaf))) ++. Node(Leaf,3,Leaf)
```

appends and applies binary tree contexts. This can be visualised as follows:



Contexts satisfy the following *context laws* [Leijen and Lorenzen 2023]

$$(\mathit{app})\ \ (\mathsf{ctx}\ \mathsf{K}) \mathbin{+\!\!+.}\ e \ = \ \mathsf{K}[e] \qquad\qquad (\mathit{dist})\ \ (c_1 \mathbin{+\!\!+} c_2) \mathbin{+\!\!+.}\ e \ = \ c_1 \mathbin{+\!\!+.}\ (c_2 \mathbin{+\!\!+.}\ e)$$

The first law states that applying a context K is the same as context substitution, where the notation $\mathsf{K}[e]$ substitutes the hole in K with e. The (dist) law states that application distributes over composition. From here, we can see that contexts with $(\mathbin{+\!\!+})$ form a monoid with the empty context as a unit. A natural implementation of contexts is as lamba expressions where context composition and application correspond to function composition and application:

$$\mathsf{ctx}\ \mathsf{K} \ = \ \lambda x.\ \mathsf{K}[x] \ \text{ with } x \notin \mathsf{fv}(\mathsf{K}) \qquad\qquad c_1 \mathbin{+\!\!+} c_2 \ = \ c_1 \circ c_2 \qquad\qquad c \mathbin{+\!\!+.}\ e \ = \ c\ e$$

We can easily check that this satisfies the context laws. We will sometimes use this naive implementation when *reasoning* about programs, but it is rather inefficient.

3.1 Minamide Tuples

Although we could implement constructor contexts using functions, the key reason for additional language support is that they can be implemented much more efficiently based on Minamide's [1998] context presentation in the linear hole calculus. In Minamide's system, a context has an affine type and it is safe to update the hole *in-place*. A context is represented by a *Minamide tuple*, written as $\{x, h\}$, where the first element x points to the top of the data structure, and the second element h points directly to the hole inside that structure. Composition and application can now directly update the hole in-place and are constant time.

Unfortunately, it is not easy to extend an existing language with Minamide's system as it requires an affine type system for contexts (and also uses linear derivations and evaluation under-lambda for contexts). In particular, this is problematic for the derivations using equational reasoning used in this paper, that do not rule out sharing or duplication of contexts.

3.2 Context Paths

There is a way to have efficient in-place mutating context operations *without* requiring affine types. The key to this is the use of *context paths*, which store the path from the root to the hole, first described by Leijen and Lorenzen [2023] in the context of tail-recursion modulo-cons optimizations in the presence of algebraic effect handlers with multi-shot resumptions. Their use of context paths is internal and not exposed to the user, but we can use the exact same runtime mechanism to implement first-class contructor contexts.

In essence, we compile constructor contexts to a runtime representation storing the context path down from the top to the hole in the data structure. To enable this, we use extra bits in the header of each object where we store the index of the child that leads to the (single) hole in the structure. Koka re-uses an 8-bit field for this purpose which is normally used for stackless freeing.

The context path indices can be constructed in constant time when compiling constructor contexts. Writing C_i for the constructor C decorated with child index i, we compile a constructor context into a Minamide tuple as follows:

$$
\begin{array}{lcl}
\mathsf{ctx}\ \Box & = & \{\square, \mathsf{NULL}\} \\
\mathsf{ctx}\ C \ldots \Box_i \ldots & = & \mathsf{let}\ x\ =\ C_i \ldots \square_i \ldots\ \mathsf{in}\ \{x,\ \&x.i\} \\
\mathsf{ctx}\ C \ldots \mathsf{K}_i \ldots & = & \mathsf{let}\ \{x, h\}\ =\ \mathsf{ctx}\ \mathsf{K}\ \mathsf{in}\ \{C_i \ldots x \ldots,\ h\} \qquad (\mathsf{K} \neq \Box)
\end{array}
$$

where $\&x.i$ denotes the address of field i in x. At runtime, a constructor allocation of C typically initializes the header fields, including the tag. Adding in the context path index yields a single constant, eliminating any overhead associated with this representation. For example, the Koka compiler compiles a context like `ctx Node(Node(Node(Leaf,1,Leaf),2,_),5,Leaf)` internally into:

```
val x = Node₃(Node(Leaf,1,Leaf),2,□) in { Node₁(x,5,Leaf), &x.3 }
```

where each constructor along the context path is annotated with a child index.

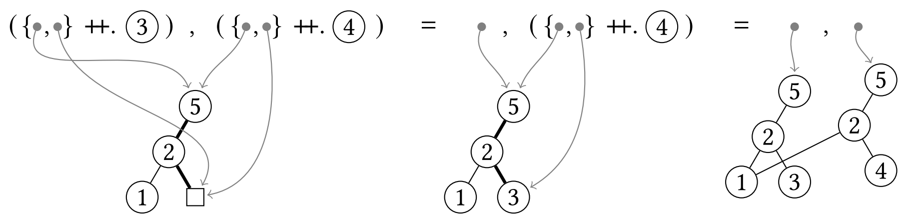
With these context paths, we can now follow the path from the top of a context to the hole in that structure *at runtime*, and thus we are able to *copy* the linear context path dynamically at runtime when required. When we compose or apply a context we can now copy shared contexts only when needed. In a language with precise reference counts (like Koka or Lean) we copy the contexts at runtime along the context paths whenever they are not unique.

We can also support this in languages without precise reference counts though. In particular, we can use a special distinguished value for a runtime hole □ that is never used by any other object. A substitution now first checks the value at the hole: if it is a □ value, the hole is substituted for the first time and we just overwrite the hole in-place (in constant time). However, any subsequent substitution on the same context will find some object instead of □. At this point, we first dynamically copy the context path (in linear time) and then update the copy in-place[2].

If the contexts happen to be used linearly, then all operations execute in constant time, just as in Minamide's approach; but we now have full functional semantics and any subsequent substitutions on the same context work correctly (but will take linear time in the length of the context path). So, the expression `val c = ctx Cons(1,_) in (c ++. [2] , c ++. [3])`, where the context `c` is shared, evaluates correctly to `([1,2],[1,3])`. Here is a more complex example of a shared tree context that

[2]It turns out that for context composition we also need to check the second context as well to avoid creating cycles, see Appendix B for details.

is applied to two separate nodes:



In this figure, the runtime context path is denoted by bold edges. The intermediate state is interesting as it is both a valid tree, but also a part of the tree is shared with the remaining context, where the hole points to a regular node now. When that context is applied, only the context path (node 5 and 2) is copied first where all other nodes stay shared (in this case, only node 1).

Just as functional languages use a combination of compilation and runtime mechanisms to implement efficient first-class lambda expressions as closures, we too use context-path compilation in combination with runtime support to implement efficient first-class constructor contexts. We retain a pure functional semantics without being restricted to linear use. Note that unlike FIP, first-class constructor contexts do not depend on precise reference counting and can be applied readily in garbage collected languages as well (see also Appendix B).

3.3 Top-Down Using Accumulating Constructor Contexts

Using the new constructor context abstraction, we are now able to define a true top-down functional version of `map` by passing down an *accumulating constructor context*:

```
fip fun map-td( xs : list<a>, ^f : a -> b, acc : ctx<list<b>> ) : list<b>
  match xs
    Cons(x,xx) -> map-td( xx, f, acc ++ ctx Cons(f(x),_) )
    Nil        -> acc ++. Nil

fip fun map(xs,f)
  map-td(xs, f, ctx _)
```

The `map` function now uses a single tail-recursive traversal down the list and returns the final list directly in constant time when reaching the end of the list (and if the list `xs` is unique at runtime it will be updated in place as well).

Similar to `map`, we can now also define a true top-down functional version of move-to-root insertion. In the recursive version of `insert` presented in Section 2.1, the recursive call always matches on the result and adds a constructor to either the left, or right tree:

```
... match insert(r,k)
      Node(s,y,b) -> Node( Node(l,x,s), y, b)
```

We can make this tail-recursive by passing *two* accumulating constructor contexts for the left- and right tree. For example, in the case outlined above, we extend the context we have accumulated so far with an additional `ctx Node(l,x,_)`. In the base cases, we then use the accumulated contexts to construct the final tree:

```
fip(1) fun insert-td( t : tree, k : key ) : tree
  down-td(t,k,ctx _, ctx _)

fip(1) fun down-td( t : tree, k : key, accl : ctx<tree>, accr : ctx<tree> ) : tree
  match t
    Node(l,x,r) -> if   x < k then down-td( r, k, accl ++ ctx Node(l,x,_), accr )
                   elif x > k then down-td( l, k, accl, accr ++ ctx Node(_,x,r) )
                   else Node( accl ++. l, x, accr ++. r )
    Leaf        -> Node( accl ++. Leaf, k, accr ++. Leaf )
```

We start with two empty contexts `ctx _`, and we can see that in each branch we extend either the left context with smaller element, or the right context with larger elements. There is no need to traverse a zipper "back up", as we saw for the bottom-up algorithm. It is straightforward to prove the following equality, relating the recursive version of move-to-root trees calculated at the beginning of this section to top-down version using constructor contexts:

Theorem 2. (*Correctness of top down move to root insert*)
For all trees `t`, keys `k` and constructor contexts `accl` and `accr`,

```
down-td(t,k,accl,accr) ≡ val Node(l,x,r) = insert(t,k) in Node(accl ++. l, x, accr ++. r)
```

Proof. The proof proceeds by induction on the tree `t`. The base case, when `t` is a leaf, is trivial. If the tree is non-empty, we distinguish three cases, depending on the key `x` stored at `t` is less than, greater than, or equal to `k`. For the first case, we need to show:

```
down-td(r, k, accl ++ Node(l,x,_), accr) ≡
val Node(l',x',r') = insert(r,k) in Node(accl ++ Node(l,x,_) ++. l', x', accr ++. r')
```

This follows from our induction hypothesis for the right subtree, where we use the (*dist*) law. The other cases are similar. As a corollary, we again obtain that the recursive version coincides with top-down version using constructor contexts:

Corollary 2. (*Top-down equals recursive specification*)
For all trees `t`, keys `k`, `insert-td(t,k) ≡ insert(t,k)`

4 FUNCTIONAL AND IMPERATIVE MOVE-TO-ROOT COINCIDE

We have now derived two functional implementations from our recursive specification. How can we relate these implementations to the imperative move-to-root algorithms published by Stephenson [1980] and Allen and Munro [1978]? As we will see, these algorithms rely heavily on pointer manipulation: it is not at all obvious that they are correct or even represent the 'same' program.

These published algorithms are usually written in imperative pseudocode. To reason about them, we formalize each algorithm precisely in Iris [Jung et al. 2018], a framework for (higher-order concurrent) separation logic [Reynolds 2002] implemented as a library in the Coq proof assistant [2017]. In the style of Frumin et al. [2019] and Bedrock2 [Erbsen et al. 2021; Pit-Claudel et al. 2022], we have defined an embedded language, called *AddressC*, building on the standard *HeapLang* language supported by Iris [2022]. The AddressC language is embedded into Coq where we use extensive `Notation` to have the embedded code resemble a low-level C-style language that can match the typical pseudo-code in published algorithms closely. Eventually, AddressC is desugared into a standard HeapLang value representing the low-level control-flow and heap operations on which the proofs operate.

While our language builds on HeapLang, we place special consideration on precisely modeling while loops and the (untyped) low-level structure of memory. For example, we model a tree as:

```
Fixpoint is_tree (t : tree) (v : val) : iProp Σ :=
  match t with
  | Leaf => ⌜v = #0⌝
  | Node l x r => ∃(p:loc) l' r', ⌜v = #p⌝ ∗ p ↦∗ [ l'; #x; r'] ∗ is_tree l l' ∗ is_tree r r'
  end.
```

This states that a `Leaf` is represented by a null address (the constant `#0`). To represent a non-empty tree, `Node l x r`, requires having some address `#p`, pointing to a heap cell of 3 fields containing an address for the left tree (`l'`), its key `#x`, and an address for the right tree `r'`. The separating conjuction, ∗, ensures that the tree is indeed inductively defined and that there are no cycles. For the bottom-up algorithms we additionally need to model zippers, which requires us to distinguish between `NodeL` and `NodeR`. To do so, we include an additional tag field in the heap cells, as `p ↦∗ [ #1; l'; #x; r']`.

Variables typically denote memory locations, and just as in C, we use `&` to take an address of a location and we use `∗` to dereference an address. We can also dereference at an offset, writing `node⟦2⟧` to dereference the second field of the `node` address. We usually define notation for constant offsets so we can write `node->right` instead of `node⟦2⟧` to get the value of the right child.

```
Definition heap_mtr_insert_td : val :=
  fun: ( name, root ) {
    var: left_dummy := #0 in
    var: right_dummy := #0 in
    var: node := root in
    var: left_hook := &left_dummy in
    var: right_hook := &right_dummy in
    while: ( true ) {
      if: ( node != #0) {
        if: ( node->value == name ) {

          *left_hook = node->left;;
          *right_hook = node->right;;
          root = node;;
          break
        }
        else {
          if: ( node->value > name )
          {
            *right_hook = node;;
            right_hook = &(node->left);;
            node = node->left
          }

          else
          {
            *left_hook = node;;
            left_hook = &(node->right);;
            node = node->right
      } } }
      else {
        *left_hook = #0;;
        *right_hook = #0;;
        root = AllocN #3 #0;;
        root->value = name;;
        break
      }
    };;
    root->left = left_dummy;;
    root->right = right_dummy;;
    ret: root
  }.
```

```
    node := root;
    left_hook := addr(left(dummy));
    right_hook := addr(right(dummy));

    while node ≠ null do
        if value(node) = name then
            begin
                0(left_hook) := left(node);
                0(right_hook) := right(node);
                root := node;
                go to bottom
            end;
        if value(node) > name then
            begin
                0(right_hook) := node;
                right_hook := addr(left(node));
                node := left(node)
            end
        else
            begin
                0(left_hook) := node;
                left_hook := addr(right(node));
                node := right(node)
            end;

    0(left_hook) := null;
    0(right_hook) := null;
    root := new_node ( );
    value(root) := name;

bottom:
    left(root) := left(dummy);
    right(root) := right(dummy)
```

Fig. 1. The move-to-root top-down algorithm formalized in AddressC on the left, versus a screenshot of Stephenson's published algorithm on the right

4.1 Proving Stephenson's Top-Down Algorithm

Stephenson [Stephenson 1980] presented an imperative top-down insertion algorithm for top-down move-to-root trees in pseudocode. Figure 1 shows both Stephenson's top-down algorithm as published and our formal AddressC implementation. We can see that our formal AddressC implementation corresponds to the published algorithm almost line-by-line. Using Iris, we can now formally relate the functional algorithm and AddressC implementation:

Theorem 3. (*Stephenson's imperative top-down move-to-root algorithm is correct*)

```
Lemma heap_mtr_insert_td_correct (k : Z) (tv : val) (t : tree) :
    {{{ is_tree t tv }}}
    heap_mtr_insert_td (ref #k) (ref tv)
    {{{ v, RET v; is_tree (mtr_insert_td k t) v }}}.
```

The pre-condition requires that the argument address `tv` points to a valid in-memory tree corresponding to `t`, and the post-condition establishes that the result address `v` points to a valid in-memory tree that corresponds to `mtr_insert_td k t`. The entire proof, relating the functional top-down move-to-root trees with their AddressC implementation, still requires about 30 lines of tactics (see Appendix D.1). The proof goes through because we can directly relate the loop invariant of this algorithm to the recursive calls of `mtr_insert_td`. As we see in the next Section, this is only possible because constructor contexts *precisely* capture the top-down behaviour of Stephenson's algorithm. It would be much harder for example to relate the AddressC code to our original recursive definition. As is often the case in verification, finding the right formulation of our theorem is vital – constructor contexts are indispensable in this proof.

4.2 Representing Constructor Contexts

Stephenson's algorithm uses intricate pointer manipulation and even goto-statements that make it non-trivial to verify formally. The key insight is that Stephenson builds the *smaller* and *bigger* trees using the `left_hook` and `right_hook` variables. For example, for the case where the key in the current node is larger than the argument key (`name`), we have:

```
if: ( node->value > name ) {
  *right_hook = node;;
  right_hook = &(node->left);;
  node = node->left }
```

Here the current node address is written to `*right_hook` which is then itself updated to point to the left child of the current node (`right_hook = &(node->left)`). Afterwards the current node is advanced to the left child. This corresponds to the functional version, where the current node is written into the right context (`accr`) and the hole is set to its left child:

```
down( l, k, accl, accr ++ ctx Node(_,x,r) )
```

At this point though, we have all kinds of problems in the formal setting. Not only have we overwritten the value that `right_hook` was previously pointing to, but we have introduced aliasing where both the current *bigger* tree's left-child and `node` point to the same location. The *bigger* tree is not even a valid constructor context as the left child is "dangling" pointer (that will eventually be overwritten). Yet we can still prove these pointer manipulations correct by relating them to the constructor contexts used in our functional algorithm. First, we define an explicit functional datatype that corresponds to the *slow path* of constructor contexts with explicit context paths:

```
type ctx0 =
  NodeL(z : ctx0, x : key, r : tree)
  NodeR(l : tree, x : key, z : ctx0)
  EndNodeL(x : key, r : tree)
  EndNodeR(l : tree, x : key)
```

```
type ctx =
  Ctx0(z : ctx0)
  Hole
```

The predicate `ctx0` relates non-empty contexts to their Minamide tuple. The `NodeL` and `NodeR` constructors represent the context path going left- or right. The `EndNodeL` and `EndNodeR` correspond to the final nodes containing the hole. Then, just as we did for `is_tree`, we define how our inductive context (`z`) corresponds to the *fast path* of a context: a Minamide tuple of two pointers to the top of the context (`v`) and the hole within (`h`).

```
Fixpoint is_ctx0 (z : ctx0) (v : val) (h : loc) : iProp Σ :=
  match z with
  | NodeL z x r  => ∃ (p : loc) z' r', ⌜v = #p⌝ ∗ p ↦∗ [ z'; #x; r'] ∗ is_ctx0 z z' h ∗ is_tree r r'
  | EndNodeL x r => ∃ (p : loc) r', ⌜v = #p⌝ ∗ ⌜h = Loc.add p 0⌝ ∗
                                     p □↦∗ 0 [ #□; #x; r' ] ∗ is_tree r r'
  ...
```

This definition ensures that `h` points to the hole in the context, and that by filling this hole, we can extend the context or re-construct the tree. The `NodeL` case is just like before. But the `EndNodeL` case is new: the `p □↦∗ 0 [ #□; #x; r' ]` predicate which states that `p` holds on to the key `#x` and subtree `r'` but not the value at position `0` where the hole is. This is different from the usual presentation [Charguéraud 2016] and allows us to change the value of the hole without inspecting the constructor context (to allow temporarily for a dangling pointer). For example, we can now prove that the following lemma holds:

```
Lemma ctx_of_ctx (z1 : ctx) (z2 : ctx0) (zv1 : loc) (h1 : loc) (zv2 : val) (h2 : loc) :
    is_ctx z1 zv1 h1 ∗ h1 ↦ zv2 ∗ is_ctx0 z2 zv2 h2 -∗ is_ctx (comp z1 z2) zv1 h2.
```

This lemma states that if the hole points to another context, together, they form the composed context. This is the key lemma that enables checking the individual cases of Stephenson's algorithm. With these definitions, the proof of the top-down algorithm is highly automated and we resolve many obligations associated with assignments on the heap automatically using diaframe [Mulder et al. 2023 2022]. The brevity of the proof – despite the intricate nature of Stephenson's algorithm – provides further evidence that these definitions capture the essence of top-down algorithms.

4.3 Proving Allen and Munro's Bottom-Up Algorithm

While Allen and Munro [1978] do not present imperative pseudo-code, we can define an imperative version of their algorithm in AddressC. They introduce a "simple exchange" (corresponding to what is now called a rotation) and describe their algorithm as:

> *[..] perform a sequence of simple exchanges on the retrieved record so that it is moved to the root. … By carefully using the coding trick of "reversing the direction of the pointers" in performing the search, only two or three extra storage locations are required.*

We can directly implement the mentioned pointer reversal technique [Schorr and Waite 1967] in AddressC (see Appendix C.1). The code corresponds closely to the functional bottom-up version we derived earlier. In particular, just as constructor contexts captured the top-down behavior in Stephenson's algorithm, a *zipper* captures the structure of in-place pointer-reversal. Even though the use of pointer-reversal is complex from a formal perspective, we can use the functional zippers to relate the functional and imperative versions to make the proofs go through.

Theorem 4. (*Allen and Munro's imperative bottom-up move-to-root algorithm is correct*)

```
Lemma heap_rebuild_correct (z : zipper) (t : root_tree) (zipper tree : loc) (zv tv : val) :
    {{{ zipper ↦ zv ∗ is_zipper z zv ∗ tree ↦ tv ∗ is_root_tree t tv }}}
    heap_rebuild #zipper #tree
    {{{ v, RET v; is_root_tree (move_to_root z t) v }}}.

Lemma heap_mtr_insert_bu_correct (i : Z) (tv : val) (t : tree) :
    {{{ is_tree t tv }}}
    heap_mtr_insert_bu (ref #i) (ref tv)
    {{{ v, RET v; is_tree (mtr_insert_bu i t) v }}}.
```

The precondition of `heap_mtr_rebuild` requires that the argument addresses, `zipper` and `tree`, point to a zipper `z` and non-leaf binary tree `t`. The postcondition guarantees that after execution, the memory location that is returned, `v`, denotes the non-leaf binary tree arising from our functional algorithm, `rebuild`. Similarly, `heap_mtr_insert_bu_correct` specifies that given an arbitrary binary tree `t` with its heap representation `tv`, the imperative version returns a tree corresponding to `mtr_insert_bu i t`.

5 SPLAY TREES

Having looked at *move-to-root* trees, we can apply the same techniques to their improved sibling, *splay* trees [Sleator and Tarjan 1985]. The move-to-root trees only move the accessed element to the root of the tree but they do not restructure the tree. As such, the tree can degrade to a list in the case of ordered accesses. Splay trees on the other hand are self-adjusting: the accessed element also restructures the path to the root to become more balanced. Sleator and Tarjan [1985] identify six different kinds of tree rotations that are required, *zig*, *zigzig*, *zigzag* and their mirrored counterparts – is it possible to derive all of these rotations?

5.1 The Essence of Splay Tree Rebalancing

We can start again with the original specification of move-to-root trees in Section 2.1 since splay trees satisfy the exact same requirements:

```
fun insert( t : tree, k : key ) : tree
  Node( smaller(t,k), k, bigger(t,k) )
```

Instead of directly deriving the recursive, top-down, and bottom-up algorithms as we did previously, we first unroll the definition of `smaller` once more:

```
fun smaller( t : tree, k : key ) : tree
  match t
    Node(l,x,r) -> if   x < k then match r
                                     Node(rl,rx,rr) ->
                                       if   x < k then Node(l,x,Node(rl,rx,smaller(rr,k)))  // (A)
                                       elif x > k then Node(l,x,smaller(rl,k)) else Node(l,x,rl)
                   elif x > k then match l
                                     Node(ll,lx,lr) ->
                                       if   x < k then Node(ll,lx,smaller(lr,k))
                                       elif x > k then smaller(lr,k) else ll
                   else l
    Leaf -> Leaf
```

Now we gain insight into why move-to-root trees can easily become unbalanced: when we move twice to the right (and dually, twice to the left for the `bigger` function) as in the branch labelled (A), we create a short unbalanced part with two right leaning nodes:

```
(Node(l,x,Node(rl,rx,rr)) -> Node(l,x,Node(rl,rx,smaller(rr,k)))   // A
```

A splay tree though *rotates* those nodes instead, resulting in a more balanced result:

```
(Node(l,x,Node(rl,rx,rr)) -> Node(Node(l,x,rl),rx,smaller(rr,k))
```

This is the essence of splay tree restructuring! It captures the key difference between move-to-root trees and splay trees. It is the only meaningful change necessary to derive splay trees, in the same style as our derivation of move-to-root trees in the previous section.

5.2 Recursive Splay Trees

We can now derive the top-down and bottom-up splay trees, as presented by Sleator and Tarjan [1985], directly from our specification. As before, we inline the unrolled definitions of `smaller` and `bigger`, and merge the branches to end up with a single recursive function:

```
fun insert( t : tree, k : key ) : tree
  match t
    Node(l,x,r) ->
      if x < k then match r
        Node(rl,rx,rr) ->
          if   rx < k then match insert(rr,k)
                             Node(rrl,rrx,rrr) -> Node(Node(Node(l,x,rl),rx,rrl),rrx,rrr)  // (A)
          elif rx > k then match insert(rl,k)
                             Node(rll,rlx,rlr) -> Node(Node(l,x,rll),rlx,Node(rlr,rx,rr))
          else Node(Node(l,x,rl),rx,rr)
        Leaf -> Node(Node(l,x,Leaf),k,Leaf)
      elif x > k then match l
        ...
      else Node(l,x,r)
    Leaf -> Node(Leaf,k,Leaf)
```

Here we marked the new restructuring case of `smaller` (A). This derived `insert` function closely mirrors the version presented by Okasaki [1999b] (Sec. 5.4).

5.3 Top-Down Splay Trees

Just as with move-to-root trees (Section 3.3), we can again use accumulating constructor contexts to build the 'smaller' and 'bigger' trees on the way down. In particular, a match on a recursive call:

```
match insert(rr,k)
  Node(rrl,rrx,rrr) -> Node(Node(Node(l,x,rl),rx,rrl),rrx,rrr)  // (A)
```

can be changed into a direct tail-recursive call, accumulating the constructor contexts:

```
splay(rr, k, accl ++ ctx Node(Node(l,x,rl),rx,_), accr)
```

The derived top-down version becomes:

```
fun insert-td( t : tree, k : key ) : tree
  down-td( t, k, ctx _, ctx _)
```

```
fip(1) fun down-td(t : tree, k : key, accl : ctx<tree>, accr : ctx<tree> ) : tree
  match t
    Node(l,x,r) ->
      if x < k then match r
        Node(rl,rx,rr) ->
          if   rx < k then down-td(rr, k, accl ++ ctx Node(Node(l,x,rl),rx,_), accr)
          elif rx > k then down-td(rl, k, accl ++ ctx Node(l,x,_), accr ++ ctx Node(_,rx,rr))
          else  Node( accl ++. Node(l,x,rl),  rx, accr ++. rr )
        Leaf -> Node( accl ++. Node(l,x,Leaf), k, accr ++. Leaf )
      elif x > k then match l
        ...
      else  Node( accl ++. l, x, accr ++. r )
    Leaf -> Node( accl ++. Leaf, k, accr ++. Leaf )
```

Now we have an efficient `fip(1)` function again. We can also formally check that top-down and direct splay tree insertion coincide:

Theorem 5. (*Correctness of top-down splay tree insertion*)

`down-td(t,k,accl,accr) ≡ val Node(l,x,r) = insert(t,k) in Node(accl ++. l, x, accr ++. r)`

5.4 Bottom-Up Splay Trees

Deriving the bottom-up version is again done by doing a CPS-transformation first, followed by defunctionalizing the closures. Since there are 4 recursive calls in the derived recursive version, the defunctionalization leads to a zipper with 4 constructors (plus `Done` for the identity):

```
type zipper
  NodeRR( l : tree, k : key, rl : tree, rk : key, up : zipper )
  NodeRL( l : tree, k : key, up : zipper, rk : key, rr : tree )
  NodeLR( ll : tree, lk : key, up : zipper, k : key, r : tree )
  NodeLL( up : zipper, lk : key, lr : tree, k : key, r : tree )
  Done
```

This type, however, is less suitable for *reuse*. Compared to the zipper used to define bottom-up move-to-root trees, we cannot reuse a `Node` constructor from our tree to extend the zipper in-place (as a node needs a $\diamond_3$ credit, but the zipper gives $\diamond_5$). Although we have lost the obvious reuse opportunity, we can recover it readily enough: since each of the nodes records *two* steps to the left or right, we can represent this with our previous zipper type. As every constructor is used after matching on two node constructors in our input tree, we can still ensure our algorithm is fully in-place. Therefore, we use the previous version of our zipper type, and replace occurrences like `NodeRL(l,x,z,rx,rr)` with `NodeL(NodeR(l,x,up),rx,rr)`. Doing so, we can once again defunctionalize the CPS transformed program to obtain a fully in-place bottom-up version of `insert`:

```
fip(1) fun insert-bu( t : tree, k : key ) : tree
  down-bu( t, k, Done)

fip(1) fun down-bu(t : tree, k : key, z : zipper ) : tree
  match t
    Node(l,x,r) ->
      if x < k then match r
        Node(rl,rx,rr) ->
          if   rx < k then down-bu(rr,k,NodeR(rl,rx,NodeR(l,x,z)))
          elif rx > k then down-bu(rl,k,NodeL(NodeR(l,x,z),rx,rr))
          else rebuild(z, Node(Node(l,x,rl),rx,rr))
        Leaf -> rebuild(z, Node(Node(l,x,Leaf),k,Leaf))
      elif ...
      else rebuild(z, Node(l,x,r))
    Leaf -> rebuild(z, Node(Leaf,k,Leaf))

fip fun rebuild( z : zipper, t : tree ) : tree
  match t
    Node(tl,tx,tr) -> match z
      NodeR(rl,rx,NodeR(l,x,up)) -> rebuild( up, Node(Node(Node(l,x,rl),rx,tl),tx,tr) ) // RR
      NodeL(NodeR(l,x,up),lx,lr) -> rebuild( up, Node(Node(l,x,tl),tx,Node(tr,lx,lr)) ) // RL
      NodeR(rl,rx,NodeL(up,x,r)) -> rebuild( up, Node(Node(rl,rx,tl),tx,Node(tr,x,r)) ) // LR
      NodeL(NodeL(up,x,r),lx,lr) -> rebuild( up, Node(tl,tx,Node(tr,lx,Node(lr,x,r))) ) // LL
      _ -> Node(tl,tx,tr)                                                                // Done
```

This version is now tail-recursive and fully in-place. We call this the *derived* bottom-up version. By induction on the tree, we can again prove that top-down and direct splay tree insertion coincide:

Theorem 6. (*Correctness of bottom-up splay tree insertion*)
`down-bu(t,k,z) ≡ rebuild(z, insert(t,k))`

5.4.1 Fused Bottom-Up. The `down-bu` function still uses nested pattern matches. Can we use a single match instead? To do so, we first need to make the recursive calls more regular by using a singleton zipper when the key is already present. In particular, we need to change a `rebuild` call like `rebuild(z, Node(Node(l,x,rl),rx,rr))` into `rebuild(NodeR(l,x,z), Node(rl,rx,rr))` instead.

When we inline `rebuild`, we can see though that this step is not exactly equivalent (in particular, when `z` is a `NodeL`). It is still giving correct binary search trees but can lead to trees that are structured slightly differently – a point we will return to in Section 5.6. For now, with this change, the nested matches become equal to the outer match and we can un-unroll them into a single match:

```
fip(1) fun down-bu-fused( t : tree, k : key, z : zipper ) : tree
  match t
    Node(l,x,r) ->
      if   x < k then down-bu-fused( r, k, NodeR(l,x,z) )
      elif x > k then down-bu-fused( l, k, NodeL(z,x,r) )
      else rebuild(z, Node(l,x,r))
    Leaf -> rebuild(z, Node(Leaf,k,Leaf))
```

All the rebalancing now takes place in the `rebuild` function which we need to extend with two more cases to handle potential singleton zippers:

```
fip fun rebuild( z : zipper, t : root ) : tree
  match root
    Root(tl,tx,tr) -> match z
      Done -> Node(tl,tx,tr)
      NodeR(rl,rx,Done) -> Node(Node(rl,rx,tl),tx,tr)                              // zig
      NodeL(Done,lx,lr) -> Node(tl,tx,Node(tr,lx,lr))
      NodeR(rl,rx,NodeR(l,x,up)) -> rebuild( up, Root(Node(Node(l,x,rl),rx,tl),tx,tr) ) // zigzig
      NodeL(NodeR(l,x,up),lx,lr) -> rebuild( up, Root(Node(l,x,tl),tx,Node(tr,lx,lr)) )
      NodeR(rl,rx,NodeL(up,x,r)) -> rebuild( up, Root(Node(rl,rx,tl),tx,Node(tr,x,r)) ) // zigzag
      NodeL(NodeL(up,x,r),lx,lr) -> rebuild( up, Root(tl,tx,Node(tr,lx,Node(lr,x,r))) )
```

We call this the *fused* bottom-up version, `insert-bu-fused`. By starting from our initial specification with just two cases for rebalancing, we now derived the usual six rebalancing cases that are common in the splay tree literature: *zig*, *zigzig*, *zigzag*, and their mirrored counterparts.

5.5 Functional and Imperative Splay Trees Coincide

Using our AddressC embedded language in Iris we formalized the published top-down and bottom-up algorithms by Sleator and Tarjan [1985], where we use pointer-reversal for bottom-up. Once again, there is a line-by-line correspondence between the published pseudocode and formal AddressC implementations that we have written (Appendix C.2 and C.3). Using the same techniques as for move-to-root trees we have formally established that the functional implementations accurately model the published imperative algorithms:

Theorem 7. (*Sleator and Tarjan's imperative top-down splay algorithm is correct*)

```
Lemma heap_splay_insert_td_correct (k : Z) (tv : val) (t : tree) :
    {{{ is_tree t tv }}}
    heap_splay_insert_td (ref #k) (ref tv)
    {{{ v, RET v; is_tree (splay_insert_td k t) v }}}.
```

Theorem 8. (*Sleator and Tarjan's imperative bottom-up splay algorithm is correct*)

```
Lemma heap_splay_insert_bu_correct (k : Z) (tv : val) (t : tree) :
    {{{ is_tree t tv }}}
    heap_splay_insert_bu (ref #k) (ref tv)
    {{{ v, RET v; is_tree (splay_insert_bu_fused k t) v }}}.
```

It is worth repeating that the proofs of these theorems are *direct*, requiring (hardly any) additional lemmas. This is only possible because the functional implementations precisely capture the iterative behaviour of their imperative counterparts through contructor contexts and zippers.

5.6 Equivalence between Splay Restructuring

Theorem 8 proves that the published imperative algorithm gives the same results as our *fused* bottom-up version – not the *derived* bottom-up one which is equivalent to the *top-down* algorithm. As also observed by Lucas [2004], the published imperative bottom-up and top-down algorithms are not equivalent.

The *fused* and *derived* algorithms have similar theoretical amortized bounds but each can create different trees. For example, if we start from a right-unbalanced tree with nodes 1 to 4 and insert node 4, we get different results for each of the various algorithms:

initial tree: move-to-root: top-down splay: (fused) bottom-up splay:

where our derived bottom-up is equivalent to the top-down splay algorithm.

The difference between the bottom-up and top-down trees may seem trivial, but it turns out our small transformation step in Section 5.4.1 may have potential theoretical consequences. In 1985, Sleator and Tarjan introduced the *dynamic optimality* conjecture which states that the cost of accesses with splay trees is within a constant factor of an optimal algorithm for performing accesses [Sleator and Tarjan 1985]. The conjecture still stands to this day, but in recent work, Levy and Tarjan [2019] present possible avenues for proving this conjecture. In particular, they show that the *subsequence* property is a sufficient (and necessary) condition for dynamic optimality. One step towards proving the subsequence property is to show that the splay algorithm has the *transformation* property where we can perform a bounded number of accesses to transform two arbitrary binary search trees with the same elements into each other. It turns out that the *fused* bottom-up algorithm has this property but unfortunately this does not hold for the published top-down algorithm (and our derived top-down and bottom-up algorithms) [Levy and Tarjan 2019].

6 ZIP TREES

In recent work, Tarjan et al. [2021] introduce *zip trees* which can be seen as the functional equivalent of *skip lists* [Pugh 1990]. A zip tree is a binary search tree where every node also has a *rank*.

```
alias key  = int
alias rank = int
type ztree
  Leaf
  Node(rank : rank, left : ztree, key : key, right : ztree)
```

We choose node ranks independently from a geometric distribution, where the rank of a node is non-negative integer k with probability $1/2^{k+1}$. Besides being max heap-ordered for the keys, the tree is now also max heap-ordered with respect to the ranks with ties broken in favor of smaller keys. We define `is-higher-rank` as:

```
fip fun is-higher-rank( ^(r1,k1) : (rank,key), ^(r2,k2) : (rank,key) ) : bool
  (r1 > r2 || (r1 == r2 && k1 < k2))
```

Any parent node always `is-higher-rank` than its children. Since a zip tree is also a binary search tree, we can also see that the rank of a parent is always greater than the rank of its left-child, and greater than or equal to the rank of its right child. Interestingly, the shape of a zip tree is now fully determined by just the rank/key pairs in the tree, and independent of the insertion order. See Figure 2 for an example of two valid zip trees. Intuitively we can see that given the geometric distribution of ranks, the shape of a tree naturally tends to be well balanced, with twice as many nodes at each lower rank. This means that the zip tree operations never need to do any explicit

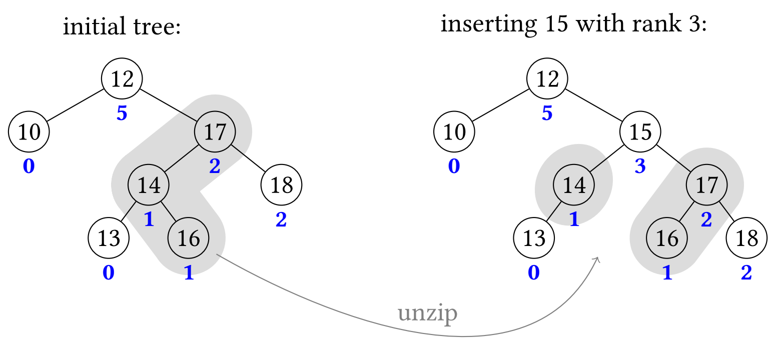


Fig. 2. Inserting a node with key 15 and rank 3 into a zip tree (with ranks shown as single digits in blue). Once the insertion point is found (as the right child of node 12), the tree at node 17 is *unzipped* along the key 15, and the resulting trees become the left- and right child of the inserted node. Deletion is the inverse where the children are *zipped* instead.

rebalancing which simplifies the implementation compared to usual balanced tree algorithms

The rank can be chosen independently at random, but in order to combine search and insertion, we can also derive the rank pseudo randomly from the key. For our implementation we define:

```
pub fip fun rank-of( ^k : key ) : rank
  val x0 = k.int32.inc
  val x1 = xor(x0,shr(x0,16))*0x297a2d39.int32
  val x2 = xor(x1,shr(x1,16))
  x2.ctz                        // count trailing zero bits
```

To insert an element into a zip tree, we first calculate the rank of the node. We can now traverse down until we find the fixed insertion point, as it is fully determined by the rank and key:

```
pub fun insert( t : ztree, k : key ) : ztree
  down( t, rank-of(k), k )

fun down( t : ztree, rank : rank, k : key ) : ztree
  match t
    Node(rnk,l,x,r) | is-higher-rank( (rnk,x), (rank,k) )  // go down while a node is higher rank
      -> if (x < k) then Node(rnk, l, x, down(r,rank,k) )
                    else Node(rnk, down(l,rank,k ), x, r)
    _ -> val (s,b) = unzip(t,k) in Node(rank,s,k,b)
```

Once we reach the insertion point where we are of higher rank than the current tree `t`, we *unzip* the tree `t` into two trees: one containing all the elements smaller than `k` and one containing all the bigger elements:

```
fun unzip( t : ztree, k : key ) : (ztree,ztree)
  (smaller(t,k), bigger(t,k))
```

Interestingly, this is almost the same initial definition of move-to-root trees. The definitions of `smaller` and `bigger` are identical to the ones we have seen previously. Figure 2 illustrates inserting a node into a tree and the resulting unzip operation. Since the shape of a zip tree is always fixed by its rank/key pairs, *deletion* is the inverse of *insertion* which *zips* child trees back together.

6.1 Recursive Zip Trees

Similar to move-to-root and splay trees, we can again inline and merge the definitions `smaller` and `bigger` and derive a direct recursive version of `unzip`:

```
fun unzip( t : ztree, k : key ) : (ztree,ztree)
  match t
    Node(rnk,l,x,r) ->
      if   (x < k) then val (s,b) = unzip(l,k) in (Node(rnk,l,x,s),b)
      elif (x > k) then val (s,b) = unzip(r,k) in (s, Node(rnk,b,x,r)
      else (l,r)
    Leaf -> (Leaf,Leaf)
```

Like before, it is straightforward to formally prove that our specification of `insert` maintains the expected properties of a zip tree.

6.2 Top-Down Zip Trees

To derive the top-down algorithm, we can again accumulate the smaller and bigger trees in constructor contexts. The `unzip` function becomes:

```
fun unzip( t : ztree, k : key, accl : ctx<ztree>, accr : ctx<ztree> ) : (ztree,ztree)
  match t
    Node(rnk,l,x,r) -> if   (x < k) then unzip( r, k, accl ++ ctx Node(rnk,l,x,_), accr )
                       elif (x > k) then unzip( l, k, accl, accr ++ ctx Node(rnk,_,x,r) )
                       else (accl ++. l, accr ++. r)
    Leaf -> (accl ++. Leaf, accr ++. Leaf)
```

Unfortunately, this is not quite `fip` since in the case that the key is present in the unzipped tree, the `else` branch discards the `Node` on which we matched. However, we can avoid calling `unzip` in the first place if the key is present and derive an efficient `fip` version:

```
fip(1) fun insert-td( t : ztree, k : key ) : ztree
  down-td( t, rank-of(k), k, ctx _)

fip(1) fun down-td( t : ztree, rank : rank, k : key, acc : ctx<ztree> ) : ztree
  match t
    Node(rnk,l,x,r) | is-higher-rank( (rnk,x), (rank,k) )
      -> if (x < k) then down-td( r, rank, k, acc ++ ctx Node(rnk,l,x,_) )
                    else down-td( l, rank, k, acc ++ ctx Node(rnk,_,x,r) )
    Node(_,_,x,_) | x == k -> acc ++. t
    _ -> val (s,b) = unzip-td( t, k, ctx _, ctx _) in acc ++. Node(rank,s,k,b)

fip fun unzip-td( t : ztree, k : key, accl : ctx<ztree>, accr : ctx<ztree> ) : (ztree,ztree)
  match t
    Node(rnk,l,x,r) -> if (x < k) then unzip-td( r, k, accl ++ ctx Node(rnk,l,x,_), accr )
                                  else unzip-td( l, k, accl, accr ++ ctx Node(rnk,_,x,r) )
    Leaf -> (accl ++. Leaf, accr ++. Leaf)
```

It is straightforward to show formally that our derivation is correct:

Theorem 9. (*Correctness of top-down zip tree insertion*)

`down-td(t,k,acc) ≡ acc ++. insert(t,k)`

where we need the following lemma for the correctness of `unzip-td`:

Lemma 1. (*Correctness of top-down unzip*)

`unzip-td(t,k,accl,accr) ≡ val (s,b) = unzip(t,k) in (accl ++. s, accr ++. b)`

6.3 Bottom-Up Zip Trees

We can also derive a bottom-up version from our recursive specification. Again, we first do a CPS conversion, and then defunctionalize the continuations into a zipper:

```
type zipper
  NodeR( rank : rank, left : ztree, key : key, up : zipper )
  NodeL( rank : rank, up : zipper, key : key, right : ztree )
  Done
```

Now we can reuse the zipper for both the `down` and `unzip` operations, as both these functions only use the zipper to "rebuild" the tree back up:

```
fip fun rebuild( z : zipper, t : ztree ) : ztree
  match z
    NodeR(rnk,l,x,up) -> rebuild(up, Node(rnk,l,x,t))
    NodeL(rnk,up,x,r) -> rebuild(up, Node(rnk,t,x,r))
    Done              -> t
```

The `down` and `unzip` now take the zipper(s) as an accumulating argument, where we again ensure we never unzip trees with the key present:

```
fip(1) fun insert-bu( t : ztree, k : key ) : ztree
  down-bu( t, rank-of(k), k, Done )
```

```
Definition heap_unzip_td : val :=
  fun: (x, key, cur) {
    var: accl := &(x->left) in    (* ctx _ *)
    var: accr := &(x->right) in
    while: (cur != #0) {
      if: (cur->key < key) {
        *accl = cur;;              (* accl ++ ctx ... Node(rnk,l,x,_) *)
        repeat: { accl = &(cur->right);; cur = cur->right }
        until: ((cur == #0) || (cur->key >= key))
      } else {
        *accr = cur;;
        repeat: { accr = &(cur->left);; cur = cur->left }
        until: ((cur == #0) || (cur->key < key))
      }
    };;
    *accl = #0;;                   (* accl ++. Leaf *)
    *accr = #0
  }.
```

if $cur = \texttt{null}$ **then** $\{x.left \leftarrow x.right \leftarrow \texttt{null}; \text{return}\}$
if $key < cur.key$ **then** $x.right \leftarrow cur$ **else** $x.left \leftarrow cur$
$prev \leftarrow x$

while $cur \neq \texttt{null}$ **do**
  $fix \leftarrow prev$
  if $cur.key < key$ **then**
    repeat $\{prev \leftarrow cur;\ cur \leftarrow cur.right\}$
    until $cur = \texttt{null}$ or $cur.key > key$
  else
    repeat $\{prev \leftarrow cur;\ cur \leftarrow cur.left\}$
    until $cur = \texttt{null}$ or $cur.key < key$
  if $fix.key > key$ or ($fix = x$ and $prev.key > key$) **then**
    $fix.left \leftarrow cur$
  else
    $fix.right \leftarrow cur$

Fig. 3. Our new formal unzip algorithm in AddressC as derived from the functional version on the left, versus a screenshot of the unzip part of Tarjan, Levy, and Timmel's algorithm on the right.

```
fip(1) fun down-bu( t : ztree, rank : rank, k : key, z : zipper ) : ztree
  match t
    Node(rnk,l,x,r) | is-higher-rank( (rnk,x), (rank,k) )
      -> if (x < k) then down-bu(r, rank, k, NodeR(rnk, l, x, z))
                    else down-bu(l, rank, k, NodeL(rnk, z, x, r))
    Node(_,_,x,_) | x == k -> rebuild(z, t)
    _ -> val (s,b) = unzip-bu(t,k,Done,Done) in rebuild( z, Node(rank,s,k,b) )

fip fun unzip-bu( t : ztree, k : key, zs : zipper, zb : zipper ) : (ztree,ztree)
  match t
    Node(rnk,l,x,r) ->
      if   (x < k) then unzip-bu( r, k, NodeR(rnk,l,x,zs), zb)
                   else unzip-bu( l, k, zs, NodeL(rnk,zb,x,r))
    Leaf -> (rebuild(zs,Leaf), rebuild(zb,Leaf))
```

We can optimize this a bit further: for the `down-bu` function, the zipper along the search path always just rebuilds the exact same path since no restructuring takes place, unlike the rebuilding for move-to-root or splay trees. It can be more efficient to use a constructor context for `down-bu` instead, as this can rebuild the tree in constant time.

For the optimized bottom-up version the correctness theorem is as before:

Theorem 10. (*Correctness of bottom-up zip tree insertion*)

`down-bu(t,k,acc) ≡ acc ++. insert(t,k)`

but now with the following lemma for the correctness of `unzip-bu`:

Lemma 2. (*Correctness of bottom-up unzip*)

`unzip-bu(t,k,zs,zb) ≡ val (s,b) = unzip(t,k) in (rebuild(zs,s), rebuild(zb,b))`

6.4 Proving Zip Trees Correct

The published imperative top-down zip insertion algorithm is interesting as it uses a minimal number of pointer assignments. However, as shown on the right side of Figure 3, it is not entirely straightforward to understand as it uses nested iterations and uses a single pointer variable *fix* to point to either the left- or right hole in each iteration. At the end of each outer iteration, we need to test whether to update the left- or right child:

if $fix.key > key$ or ($fix = x$ and $prev.key > key$) **then**
  $fix.left \leftarrow cur$
else
  $fix.right \leftarrow cur$

This test complicates the algorithm since it resolves differently in the first iteration (where $fix = x$) than subsequent ones. But perhaps we can avoid such checks in the first-place?

What we can do instead is "derive" an imperative algorithm from our functional one by manually "compiling" to AddressC code and removing any checks and code that deal with reference counting and handling of shared data. The listing on the left in Figure 3 shows the AddressC code that we can derive this way from our functional top-down `unzip-td` function (in Section 6.2), next to a screenshot of the unzip part of the algorithm by Tarjan, Levy, and Timmel [2021, Algorithm 2]. Our derived algorithm uses two accumulator contexts, `accl` and `accr`, instead of a single *fix* variable, and there is no need for an extra test at the end of each iteration. If we translate directly from our functional `unzip-td`, a context composition such as `accl ++ ctx Node(rnk,l,x,_)` would actually become:

```
*accl = cur;;          (* accl ++ ctx Node(rnk,l,x,_) *)
accl  = &cur->right;;
cur   = cur->right     (* tail-call *)
```

without an inner repeat-until loop. However, while we traverse right children, where `cur->key < key`, we would now update the right-child hole with same right tree address on each iteration! To minimize the number of pointer assignments, we can instead construct a *larger* context as a chain of right-child nodes as long as `cur->key < key`. In our algorithm in Figure 3 we use a nested iteration to move the hole as far as possible along the right children:

```
*accl = cur;;          (* accl ++ ctx Node(rnk1,l1,x1, ... Node(rnkN,lN,xN, _) ...) *)
repeat: {
  accl = &cur->right;;
  cur = cur->right
} until: ((cur == #0) || (cur->key >= key))
```

This is an optimization that we cannot directly express on the functional side at this time. Gudjónsson and Winsborough [1999] have already studied a similar optimization in their work on compile-time reuse in Prolog. As another example, the same situation occurs in the ubiqutious `map` function (see Section 3.3): if all nodes in the mapped list are reused, the tail of each `Cons` is overwritten with the same tail address.

Just like the published algorithm by Tarjan, Levy, and Timmel, our final derived algorithm (shown fully in Appendix C.4) now uses minimal pointer assignments, but it is shorter with fewer tests and branches, and we can prove it correct:

Theorem 11. (*Imperative top-down zip tree insertion is correct*)

```
Lemma heap_zip_insert_td_correct (k rank : Z) (tv : val) (t : ziptree) :
    {{{ is_ziptree t tv }}}
    heap_zip_insert_td (ref tv) (ref #rank) (ref #k)
    {{{ v, RET v; is_ziptree (zip_down_td t rank k Hole) v }}}.
```

There is no published bottom-up algorithm, but just as with the top-down version we can easily derive an efficient bottom-up algorithm in AddressC from our functional version (Appendix C.5) as well and prove this correct:

Theorem 12. (*Imperative bottom-up zip tree insertion is correct*)

```
Lemma heap_zip_insert_bu_correct (tv : val) (t : ziptree) (rank : Z) (k : Z) (zv : val) (z : zipper) :
    {{{ is_ziptree t tv ∗ is_zipper z zv }}}
    heap_zip_insert_bu (ref tv) (ref #rank) (ref #k) (ref zv)
    {{{ v, RET v; is_ziptree (zip_down_bu t rank k z) v }}}.
```

7 BENCHMARKS

Figure 4 shows benchmark results for the various derived algorithms in this paper. We compare Koka against the best known iterative C implementations. For bottom-up algorithms, we also benchmark ML and Haskell implementations that are direct translations of the bottom-up Koka versions. We ran the benchmarks on Ubuntu 22.04.2 using an AMD 7950X at 4.5Ghz. We used Koka v2.4.2 (`-O2`), the C implementations were compiled with Clang 14 (`-O3 -DNDEBUG`), ML with OCaml 4.13.1 (`ocamlopt -O2`), and Haskell with GHC 8.8.4 (`-O2`). Each benchmark performs 10M insertions starting with an empty tree, using a pseudo random sequence of keys between 0 and 100 000. Initially the tree is populated quickly up to 100 000 elements followed by many insertions where the element already exists. We tested all top-down (*-td*) and bottom-up (*-bu*) versions of move-to-root tree (*mtr*), splay trees (*splay*), and zip trees (*zip*). Figure 4 also includes tests for red-black trees (*rb*) but we disregard those for the moment.

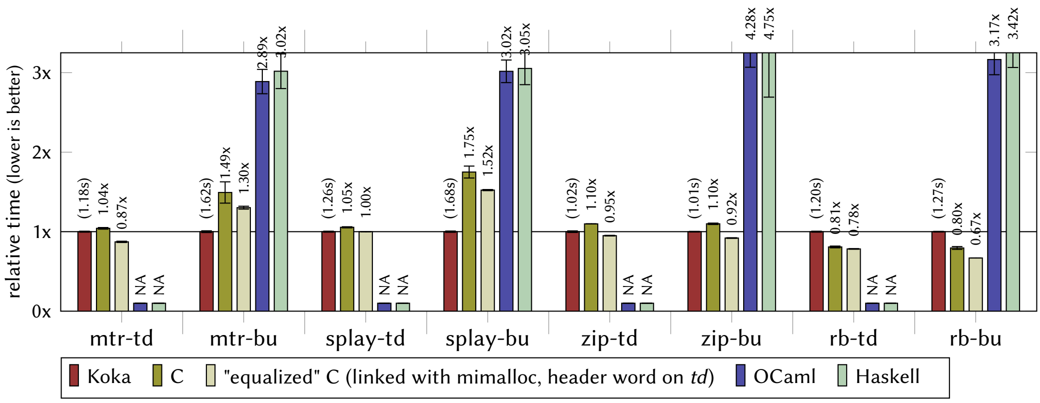


Fig. 4. Benchmarks on Ubuntu 22.04.2 (AMD 7950X 4.5Ghz) comparing the relative performance of C, ML, and Haskell against Koka for move-to-root (*mtr*), splay trees (*splay*), and zip trees (*zip*) for both top-down (*td*) and bottom-up (*bu*) variants. Each benchmark performs the same sequence of 10M pseudo-random insertions between 0 and 100 000 starting with an empty tree.

If we look at the performance relative to Koka in Figure 4 we see that our purely functional *fip* derived versions always outperform C for move-to-root, splay, and zip-trees! How is that even possible? The Koka code in particular must perform more operations:

- Koka has automatic memory management, and thus everything is reference counted. The generated code also includes branches to handle potential thread shared structures (which requires atomic reference count operations).
- Koka uses arbitrary precision integers (`int`) for keys and all comparisons and arithmetic operations include branches for the case where big integer arithmetic is required.
- Context composition and application are reference counted to handle sharing, and always check for empty contexts. In the C code empty context checks are unnecessary due to stack allocation.
- Koka reuses memory when possible, but the trees can always be used *persistently* as well, and insertion can also handle shared trees where the spine to the insertion point is copied.

One factor why Koka still outperforms C is that Koka is tightly integrated with the optimized *mimalloc* allocator [Leijen et al. 2019]. To gain better insight into what the actual overheads of the above features are in our functional code, we also include "equalized" C: here we link the C programs with *mimalloc* as well (overriding `malloc` and `free`), and we include an unused header word in the top-down algorithms to ensure an equal amount of memory is allocated.[3] This is the third bar in Figure 4. Even compared to equalized C our functional versions still perform remarkably well, being at most 15% slower for top-down move-to-root trees, and only 6% slower for top-down zip-trees. This is suprising, given the additional safety guarantees Koka provides. Many of these checks are *cache-local* and use just few instructions in the fast path (e.g. `is-unique`). We conjecture that on modern hardware small fast-path branches with cache-local accesses can be quite cheap – due to the speculation with many parallel compute units the actual performance bottlenecks may be somewhere else, such as a dependency on an uncached memory read.

Even with equalized C, our functional versions are still substantially faster on the bottom-up move-to-root and splay trees. This is due to the difference in implementations: in our derived functional versions we use zippers which are compiled essentially to use in-place pointer-reversal at runtime. The C implementations, in constrast, are using parent pointers instead which is the usual way of traversing back up for the bottom-up algorithms. However, for move-to-root and

[3] This is not required for the bottom-up algorithms in C since these have parent pointers which balances out against Koka's header words (which uses pointer reversal through zippers and needs no parent pointers).

splay trees the constant restructuring is now more expensive since we need to also adjust parent pointers for each rotation. This cost is much less pronounced for zip trees for which considerably less restructuring takes place, and so the performance difference is correspondingly smaller. As an experiment, we also implemented a pointer-reversal version of Allen and Munro's move-to-root bottom-up algorithm using the lowest pointer bit to distinguish left- from right paths. In that case, the equalized C code performs about 14% better than our functional version.

The top-down zip tree algorithm in C uses Tarjan et al.'s algorithm. We also tested this with our derived algorithm, and the simpler version that does not have the inner *repeat-until* iterations (and may thus perform extra pointer assignments as shown in Section 6.4). For our benchmark though, we could not measure any significant differences in execution times between these versions.

Red-Black Trees. Figure 4 also contains benchmark results for bottom-up [Guibas and Sedgewick 1978; Lorenzen and Leijen 2022; Okasaki 1999a] and top-down [Tarjan 1985; Weiss 2013] red-black tree algorithms. It is beyond the scope of this paper to discuss those in detail but we can apply the same techniques that we have shown in this paper to implement the bottom-up version using defunctionalized CPS and zippers, and the top-down version with constructor contexts. The top-down C version is based on the GNU library tree search implementation which encodes the node color in the least significant pointer bit [Schmidt 1997], while the bottom-up one implements the algorithm described by Cormen et al. [2022]. The relative performance of Koka versus (equalized) C is still good, but less impressive as for the other data structures: about 25% slower for the top-down algorithm and almost 50% slower for the bottom-up version. This shows that there is still room for further improvements in our compilation techniques.

Each variant performs slower for different reasons though. We believe the functional version of bottom-up red-black trees is slower because the C versions can use early bailout: on the way back up as soon as a parent is no longer red, the C version can immediately return the root pointer. For the functional version though we need to unwind the zipper completely to reconstruct the tree. There seems no obvious way to implement such optimization on the functional side – we would need some concept of parent pointers to achieve similar behaviour. For the top-down version the slower performance is less clear, but we believe it is due to the need to keep track of extra context. Top-down red-black tree rebalancing requires access to the parent and grand-parent of the current node for its rebalancing operations. In C we can just keep two extra pointers around on the traversal down. In the functional version though we need two derivative node constructors for the parent and grandparent, together with the accumulating constructor context – moving the grand-parent into the constructor context on each iteration. We imagine that a potential path to improving this situation is to allow a limited form of pattern matching on constructor contexts.

8 RELATED WORK

We discuss related work of the studied algorithms in the main text. Here, we want to present an overview of the work related to the employed techniques.

Datastructures with a hole. Zippers [Huet 1997] are the canonical functional representation of datastructures with a hole. They can be derived from types [Hinze et al. 2004; McBride 2001 2008], but also arise syntactically as the defunctionalization of the closures generated by a CPS-conversion [Danvy and Nielsen 2001]. While they have long been known to be the functional equivalent of backpointers [Huet 2003; Munch-Maccagnoni and Douence 2019], only recently has this insight been exploited to actually compile them to pointer reversing code [Lorenzen et al. 2023b].

In contrast, constructor contexts, as studied in this work, have received far less attention. One reason for this may be that previous implementations required type systems to ensure safety. Minamide [1998] describes a linear type system for efficient one-hole contexts, while destination passing style [Bour et al. 2021; Pottier and Protzenko 2013] requires linear or ownership types.

Huet [2003] also discusses top-down structures with a hole Ω, but he does not make an explicit connection to top-down algorithms or present an efficient implementation.

Some top-down algorithms can also be expressed using either laziness [Wadler 1984] or tail recursion modulo cons (TRMC) [Bour et al. 2021; Friedman and Wise 1975; Leijen and Lorenzen 2023; Risch 1973]. However, both techniques require the programmer to provide an expression up-front which determines the value eventually filling the hole. This makes it more cumbersome and sometimes impossible to express top-down algorithms with these techniques. Laziness additionally carries a performance overhead due to the creation of intermediate thunks. Conversely, TRMC can be implemented manually with first-class constructor contexts: Leijen and Lorenzen [2023] introduce the *context transformation*, which generalises Danvy and Nielsen's [2001] approach to constructor contexts.

Compilation of functional programs. A crucial step of our compilation is to *reuse* [Lorenzen and Leijen 2022; Schulte and Grieskamp 1992; Ullrich and de Moura 2019] old heap cells for new ones. This can be performed automatically in languages with precise reference counting [Reinking, Xie et al. 2021; Ullrich and de Moura 2019], but could also be manually implemented in languages with uniqueness types [Barendsen and Smetsers 1996]. However, in order to achieve a fully in-place algorithm, we also need to be sure that certain values (such as tuples) are not allocated on the heap. Lorenzen et al. [2023b] propose a calculus for such functions which ensures that the functions presented here do not have spurious allocations.

In this work, we study compilation as a refinement [Appel 2016] which allows us to connect the functional implementation to published imperative code. Modulo exact choice of variable names and helper functions, it is possible to *compile* functional code directly to published imperative code. Hofmann [2000] first proposed such a scheme and Gudjónsson and Winsborough [1999] presents an optimization to avoid updating the hole of the context in cases where it already contains the right value, just as in the published implementation of zip tree insertion.

Verification of imperative algorithms. Insertion and deletion algorithms for binary search trees have been verified countless times: There is a large literature on functional implementations [Nipkow et al. 2021 2020] as well as destructive implementations [Armborst and Huisman 2021; Pek et al. 2014; Stefanescu et al. 2016 2016; Zhan 2018]. However, these algorithms are typically based on *recursive* code and thus do not deal with the issues discussed in this paper. Surprisingly, there seems to be far less literature on verifying idiomatic, imperative code as it appears in algorithm papers. Schellhorn et al. [2022] and Dross and Moy [2017] formalize the text-book insertion and deletion of red-black trees, but due to the use of inline invariants their code does not resemble the original implementation. Lammich [2020] formalizes an array-based implementation of pattern-defeating quicksort in the Boost C++ library. Enea et al. [2015] prove insertion algorithms for AVL trees and red-black trees in C correct by deriving a representation for the already-traversed segment. They do not consider a functional version and thus have to perform a proof search.

Formalizing constructor contexts. Following Charguéraud [2016], we define an inductive representation of one-hole datastructures. In the work of Enea et al. [2015], these *segments* also hold additional invariants, but this is not necessary if one only wants to relate the segments to their functional counterpart. Cao et al. [2019] formalize an idiomatic, non-balancing insertion into binary trees and point out that a constructor context can also be represented in separation logic using a magic wand, thereby hiding the implementation of constructor contexts behind their interface. This representation avoids having to define segments and opens an opportunity to re-create this work in tools that have special support for magic wands such as Viper [Dardinier et al. 2022; Müller et al. 2016]. Tuerk [2010] demonstrates a method for proving the correctness of while-loops that recurse on an argument by using simple pre- and post-conditions; this may be powerful enough to prove the correctness of bottom-up algorithms as well as those top-down algorithms that arise from a functional version via TRMC.

9 CONCLUSION AND FUTURE WORK

We believe that the techniques presented in this paper can be applied to other algorithms and datastructures. It may be interesting to explore and relate the different implementations of datastructures such as red-black trees from a functional perspective. It may also be possible to derive efficient imperative implementations of traditional functional datastructures, that are not yet well known in the imperative world. Lorenzen et al. [2023b] already describe such an implementation for finger trees.

REFERENCES

Adams. 1993. Functional Pearls Efficient Sets—a Balancing Act. *Journal of Functional Programming* 3 (4). Cambridge University Press: 553–561. doi:https://doi.org/10.1017/S0956796800000885.

Ager, Biernacki, Danvy, and Midtgaard. 2003. A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, 8–19. PPDP '03. Association for Computing Machinery, New York, NY, USA. doi:https://doi.org/10.1145/888251.888254.

Allen, and Munro. 1978. Self-Organizing Binary Search Trees. *Journal of the ACM (JACM)* 25 (4). ACM New York, NY, USA: 526–535.

Appel. 2016. Modular Verification for Computer Security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 1–8. IEEE.

Armborst, and Huisman. 2021. Permission-Based Verification of Red-Black Trees and Their Merging. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, 111–123. IEEE.

Aspinall, Hofmann, and Konečný 2008. A Type System with Usage Aspects. *Journal of Functional Programming* 18 (2). Cambridge University Press: 141–178.

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

Bell, Bellegarde, and Hook. 1997. Type-Driven Defunctionalization, ICFP '97, . Association for Computing Machinery, New York, NY, USA, 25–37. doi:https://doi.org/10.1145/258948.258953.

Blelloch, Ferizovic, and Sun. 2016. Just Join for Parallel Ordered Sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 253–264.

Bour, Clément, and Scherer. Apr. 2021. Tail Modulo Cons. *Journeées Francophones Des Langages Applicatifs (JFLA)*, April. Saint Médard d'Excideuil, France. `https://hal.inria.fr/hal-03146495/document`. hal-03146495.

Cao, Wang, Hobor, and Appel. 2019. Proof Pearl: Magic Wand as Frame. *arXiv Preprint arXiv:1909.08789*.

Charguéraud. 2016. Higher-Order Representation Predicates in Separation Logic. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, 3–14.

Clark, and Tärnlund. 1977. A First Order Theory of Data and Programs. In *IFIP Congress*, 939–944.

Cormen, Leiserson, Rivest, and Stein. 2022. *Introduction to Algorithms*. MIT press.

Danvy, Millikin, and Nielsen. Nov. 2007. On One-Pass CPS Transformations. *J. Funct. Program.* 17 (6): 793–812. doi:https://doi.org/10.1017/S0956796807006387.

Danvy, and Nielsen. 2001. Defunctionalization at Work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 162–174.

Dardinier, Parthasarathy, Weeks, Müller, and Summers. 2022. Sound Automation of Magic Wands. In *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II*, 130–151. Springer.

Dross, and Moy. 2017. Auto-Active Proof of Red-Black Trees in SPARK. In *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings 9*, 68–83. Springer.

Enea, Sighireanu, and Wu. 2015. On Automated Lemma Generation for Separation Logic with Inductive Definitions. In *Automated Technology for Verification and Analysis: 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings 13*, 80–96. Springer.

Erbsen, Gruetter, Choi, Wood, and Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 604–619.

Friedman, and Wise. Dec. 1975. *Unwinding Stylized Recursion into Iterations*. 19. Bloomingdale, Indiana. `https://legacy.cs.indiana.edu/ftp/techreports/TR19.pdf`.

Frumin, Gondelman, and Krebbers. 2019. Semi-Automated Reasoning About Non-Determinism in C Expressions. In *ESOP*, 60–87.

Gudjónsson, and Winsborough. 1999. Compile-Time Memory Reuse in Logic Programming Languages through Update in Place. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21 (3). ACM New York, NY, USA: 430–501.

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

Hinze, Jeuring, and Löh. 2004. Type-Indexed Data Types. *Science of Computer Programming* 51 (1-2). Elsevier: 117–151.

Hofmann. 2000. In-Place Update with Linear Types or How to Compile Functional Programms into Malloc-Free C. *Preprint, Www. Dcs. Ed. Ac. Uk/~ Mxh/malloc. Ps. Gz.* Citeseer.

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

Huet. 2003. Linear Contexts, Sharing Functors: Techniques for Symbolic Computation. *Thirty Five Years of Automating Mathematics*. Springer, 49–69.

Hughes. 1986. A Novel Representation of Lists and Its Application to the Function "reverse." *Information Processing Letters* 22 (3). Elsevier: 141–144.

Jung, Krebbers, Jourdan, Bizjak, Birkedal, and Dreyer. 2018. Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28. Cambridge University Press: e20. doi:https://doi.org/10.1017/S0956796818000151.

Lammich. 2020. Efficient Verified Implementation of Introsort and Pdqsort. In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II 10*, 307–323. Springer.

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

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

Leijen, and Lorenzen. Jul. 2022. *Tail Recursion Modulo Context – An Equational Approach*. MSR-TR-2022-18. Microsoft Research.

Leijen, and Lorenzen. Jan. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proc. ACM Program. Lang.* 7 (POPL). doi:https://doi.org/10.1145/3571233. See also [Leijen and Lorenzen 2022].

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

Levy, and Tarjan. 2019. A New Path from Splay to Dynamic Optimality. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1311–1330. SODA '19. Society for Industrial and Applied Mathematics, San Diego, California.

Lorenzen, and Leijen. Sep. 2022. Reference Counting with Frame Limited Reuse. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming (ICFP'2022)*. ICFP'22. Ljubljana, Slovenia.

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

Lorenzen, Leijen, and Swierstra. Sep. 2023. FP^2: Fully in-Place Functional Programming. In *Proceedings of the 28th ACM SIGPLAN International Conference on Functional Programming (ICFP'2023)*. ICFP'23. Seattle,USA. Under submission. See [Lorenzen et al. 2023a] for the extended technical report.

Lucas. May 2004. A Direct Algorithm for Restricted Rotation Distance. *Inf. Process. Lett.* 90 (3). Elsevier North-Holland, Inc.: 129–134. doi:https://doi.org/10.1016/j.ipl.2004.02.001.

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

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

Minamide. 1998. A Functional Representation of Data Structures with a Hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 75–84. POPL '98. San Diego, California, USA. doi:https://doi.org/10.1145/268946.268953.

Mulder, Czajka, and Krebbers. 2023. Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic. *Proceedings of the ACM on Programming Languages* 7 (PLDI). ACM New York, NY, USA: 1340–1364.

Mulder, Krebbers, and Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 809–824.

Munch-Maccagnoni, and Douence. 2019. Efficient Deconstruction with Typed Pointer Reversal. In *ML 2019-Workshop*, 1–8.

Müller, Schwerhoff, and Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, 41–62. Springer.

Nipkow, Blanchette, Eberl, Gómez-Londoño, Lammich, Sternagel, Wimmer, and Zhan. 2021. Functional Algorithms, Verified.

Nipkow, Eberl, and Haslbeck. 2020. Verified Textbook Algorithms: A Biased Survey. In *Automated Technology for Verification and Analysis: 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19–23, 2020, Proceedings 18*, 25–53. Springer.

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

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

Pek, Qiu, and Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. *ACM SIGPLAN Notices* 49 (6). ACM New York, NY, USA: 440–451.

Pit-Claudel, Philipoom, Jamner, Erbsen, and Chlipala. 2022. Relational Compilation for Performance-Critical Applications: Extensible Proof-Producing Translation of Functional Models into Low-Level Code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 918–933.

Plotkin. 1975. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science* 1 (2): 125–159. doi:https://doi.org/10.1016/0304-3975(75)90017-1.

Pottier, and Protzenko. 2013. Programming with Permissions in Mezzo. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 173–184. ICFP '13. ACM, Boston, Massachusetts, USA. doi:https://doi.org/10.1145/2500365.2500598.

Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM* 33 (6). ACM New York, NY, USA: 668–676.

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

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

Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 55–74. IEEE.

Risch. Nov. 1973. REMREC - A Program for Automatic Recursion Removal. Inst. för Informationsbehandling, Uppsala Universitet. `https://user.it.uu.se/~torer/publ/remrec.pdf`.

Schellhorn, Bodenmüller, Bitterlich, and Reif. 2022. Separating Separation Logic–modular Verification of Red-Black Trees. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, 129–147. Springer.

Schmidt. 1997. The GNU C Library, `tsearch`. `https://github.com/lattera/glibc/blob/master/misc/tsearch.c`.

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

Schulte, and Grieskamp. 1992. Generating Efficient Portable Code for a Strict Applicative Language. In *Declarative Programming, Sasbachwalden 1991*, 239–252. Springer.

Sleator, and Tarjan. 1985. Self-Adjusting Binary Search Trees. *Journal of the ACM (JACM)* 32 (3). ACM New York, NY, USA: 652–686.

Stefanescu, Park, Yuwen, Li, and Roşu. 2016. Semantics-Based Program Verifiers for All Languages. *ACM SIGPLAN Notices* 51 (10). ACM New York, NY, USA: 74–91.

Stephenson. 1980. A Method for Constructing Binary Search Trees by Making Insertions at the Root. *International Journal of Computer & Information Sciences* 9. Springer: 15–29.

Tarjan. May 1985. *Efficient Top-Down Updating of Red-Black Trees*. TR-006-85. Princeton University. `https://www.cs.princeton.edu/research/techreps/TR-006-85`.

Tarjan, Levy, and Timmel. 2021. Zip Trees. *ACM Transactions on Algorithms (TALG)* 17 (4). ACM New York, NY: 1–12.

The Coq Development Team. Oct. 2017. *The Coq Proof Assistant, Version 8.7.0* (version 8.7.0). Zenodo. doi:https://doi.org/10.5281/zenodo.1028037.

The Iris Team. 2022. *The Iris 4.0 Reference*. `https://plv.mpi-sws.org/iris`.

Tuerk. 2010. Local Reasoning about While-Loops. *VSTTE* 2010: 29.

Ullrich, and de Moura. Sep. 2019. Counting Immutable Beans – Reference Counting Optimized for Purely Functional Programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL'19)*. Singapore.

Wadler. 1984. Listlessness Is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-Time. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 45–52.

Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 61–78.

Weiss. 2013. *Data Structures and Algorithm Analysis in C++ (Fourth Edition)*. Addison-Wesley.

Zhan. 2018. Efficient Verification of Imperative Programs Using auto2. In *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24*, 23–40. Springer.

A PROGRAMMING WITH CONSTRUCTOR CONTEXTS

As also shown by Minamide [1998], there are various standard functions that can be implemented more efficiently using constructor contexts. We already saw the top-down version of `map` in Section 3.3:

```
fip fun map-td( xs : list<a>, ^f : a -> b, acc : ctx<list<b>> ) : list<b>
  match xs
    Cons(x,xx) -> map-td( xx, f, acc ++ ctx Cons(f(x),_) )
    Nil        -> acc ++. Nil

fip fun map(xs,f)
  map-td(xs, f, ctx _)
```

The `map` function is actually tail-recursive *modulo cons* [Bour et al. 2021; Friedman and Wise 1975; Leijen and Lorenzen 2023; Risch 1973], and can potentially be optimized by a compiler automatically to a form that mimics `map-td`. The scope of TRMC optimizations is limited though, and with first-class contexts we can go beyond that. Consider the `flatten` function which concatenates a list of lists, and is usually defined as:

```
fun flatten( xss : list<list<a>> ) : list<a>
  match xss
    Cons(xs,xxs) -> append( xs, flatten(xxs) )
    Nil          -> Nil
```

The `flatten` function is not tail-recursive modulo cons, and uses stack space linear in the size of the input list. Again, we can use an accumulating constructor context to flatten the lists in one traversal. Key to this is the ability to return the accumulator as a first-class result value from `append` instead of applying it directly[4]:

```
fip fun append-td( xs : list<a>, acc : ctx<list<a>> ) : ctx<list<a>>
  match xs
    Cons(x,xx) -> append-td( xx, acc ++ ctx Cons(x,_) )
    Nil        -> acc

fbip fun flatten-td( xss : list<list<a>>, acc : ctx<list<a>> ) : ctx<list<a>>
  match xss
    Cons(xs,xxs) -> flatten-td( xxs, append-td( xs, acc ) )
    Nil          -> acc

fbip fun flatten( xss : list<list<a>> ) : list<a>
  flatten-td( xss, ctx _) ++. Nil
```

Since constructor contexts are first-class, we can return them from functions like `append-td` and also store them as intermediate results. In the case of lists, they are an efficient implementation of difference lists [Clark and Tärnlund 1977; Hughes 1986] and similar techniques can be used for functions like `filter`, `partition`, `zip`, etc.

A.1 Union on Zip Trees

As another interesting example of the usefulness of first-class constructor contexts, we take a look at the union operation on zip trees. A common imperative approach is to use an intermediate array, but we would like to do this in an in-place divide-and-conquer style for optimal efficiency [Adams 1993; Blelloch et al. 2016]. To do this we define a variant of the top-down `find`, which we call `split`. This splits a tree at the insertion point for a key into three parts: the tree above the insertion point as a context, and the unzipped smaller and bigger tree:

[4]The `flatten` function is `fbip` (instead of `fip`) as it deallocates the `Cons` nodes of the outer list.

```
fip(1) fun split( t : ztree, rank : rank, k : key, acc : ctx<ztree> ) : (ctx<ztree>,ztree,ztree)
  match t
    Node(rnk,l,x,r) | is-higher-rank( (rnk,x), (rank,k) )
          -> if x < k  then split(r, rank, k, acc ++ ctx Node(rnk,l,x,_))
                       else split(l, rank, k, acc ++ ctx Node(rnk,_,x,r))
    Node(_,l,x,r)
          -> if x == k then (acc,l,r)
                       else val (s,b) = unzip(t, k, ctx _, ctx _) in (acc,s,b)
    Leaf  -> (acc,Leaf,Leaf)
```

Note that we cannot quite (re)use this function for general insertion as it may deallocate a single node if the key is already present (and for insertion we want to reuse such a node in-place and thus need the specialized `down` function). Here we return the constructor context of the tree above the insertion as a first-class result. We can now write an efficient in-place `union` function:

```
fbip fun union( t1 : ztree, t2 : ztree ) : ztree
  match t1
    Node(rnk,l1,x,r1) ->
      val (top,l2,r2) = split( t2, rnk, x, ctx _ )
      top ++. Node(rnk, union(l1,l2), x, union(r1,r2))
    Leaf -> t2
```

Here we use `split` to split the second tree around the insertion point for `x`. Due to the fixed shape of a zip tree (and having the rank being determined by the key), the new node is always of higher rank than `l1,l2` and `r1,r2`, and must come under `top` – and we can recursively construct the left- and right tree as the union of `l1,l2` and `r1,r2` respectively. The `union` function is marked `fbip` [Lorenzen et al. 2023b] as it does not allocate – but it is not quite `fip` as it may deallocate nodes that are in both trees, and needs stack space linear in the depth of the first tree.

```c
struct ctx_t {              // a Minamide context
  heap_block_t*   root;
  heap_block_t**  hole;
};

struct ctx_t  ctx_copy( struct ctx_t c ) {
  ... // copy `c.root` along the context path until reaching `c.hole`
}

// (++.) : cctx<a,b> -> b -> a
heap_block_t*  ctx_apply( struct ctx_t c1, heap_block_t* x )
{
  // is c1 an empty context?
  if (c1.root == NULL) return x;

  // copy c1 ?
  struct ctx_t  d1 = (*c1.hole != HOLE ?  ctx_copy(c1) : c1);                          // (A)

  *d1.hole = x;
  return d1.root;
}

// (++) : cctx<a,b> -> cctx<b,c> -> cctx<a,c>
struct ctx_t  ctx_compose( struct ctx_t c1, struct ctx_t c2 )
{
  // is c1 an empty context?
  if (c1.root == NULL) return c2;

  // copy c1 ?
  struct ctx_t  d1 = (*c1.hole != HOLE  ?  ctx_copy(c1) : c1 );                        // (A)

  // copy c2 ? (needed to avoid cycles)
  struct ctx_t  d2 = ((*c2.hole != HOLE || c1.hole == c2.hole) ? ctx_copy(c2) : c2 );  // (B)

  *d1.hole = d2.root;
  d1.hole  = d2.hole;
  return d1;
}
```

Fig. 5. Implementing constructor composition and application in the runtime system (for languages without precise reference counts).

B IMPLEMENTING CONSTRUCTOR CONTEXTS

Figure 5 shows a partial implemention in C code of how one can implement constructor contexts in a runtime for languages without precise reference counting. We assume that `HOLE` is the distinguished value for unfilled holes (□). When we compose two contexts we need to ensure we can handle shared contexts as well where we copy a context along the context path if needed (using `ctx_copy`).

In the application and composition functions, the check `(A)` sees if the hole in `c1` is already overwritten (where `*c1.hole != HOLE`). In that case we copy `c1` along the context path as shown in Section 3.2 to maintain referential transparency.

However, in the composition operation we also need to do a similar check for `c2` as well in order to avoid cycles: the second check `(B)` checks if `c2` has an already overwritten hole, but also if the hole in `c2` is the same as in `c1`. In either case, `c2` is copied along the context path. Effectively, both checks ensure that the new context that is returned always ends with a single fresh `HOLE`. Let's consider some examples of shared contexts. A basic example is a simple shared context, as in:

```
val c = ctx Cons(1,_) in (c ++. [2], c ++. [3])
```

which evaluates to `([1,2],[1,3])`. Here, during the second application, check `(A)` ensures the shared context `c` is copied such that the list `[1,2]` stays unaffected.

A more tricky example is composing a context with itself:

```
val c = ctx Cons(1,_) in (c ++ c) ++. [2]
```

which evaluates to `[1,1,2]`. The check `(B)` here copies the appended `c` (since `c1.hole == c2.hole`). In this example the potential for a cycle is immediate, but generally it can be obscured with a shared context inside another. Consider:

```
val c1 = ctx Cons(1,_)
val c2 = ctx Cons(2,_)
val c3 = ctx Cons(3,_)
val c  = c1 ++ c2 ++ c3 in  (c ++ c2) ++. [4]
```

which evaluates to `[1,2,3,2,4]`. The check `(B)` again copies the appended `c2` in `c ++ c2` (since `*c2.hole != HOLE`).

Note that the (B) check in composition is sufficient to avoid cycles. In order to create a cycle in the context path, either `c1` must be in the context path of `c2` (I), or the `c2` in the context path of `c1` (II). For case (I), if `c1` is at the end of `c2`, then their holes are at the same address where `c1.hole == c2.hole`. Otherwise, if `c1` is not at the end, then `*c1.hole != HOLE` and we have copied `c1` already due to check (A). For case (II) the argument is similar: if `c2` is at the end of `c1` we again have `c1.hole == c2.hole`, and otherwise `*c2.hole != HOLE`.

Languages with Precise Reference Counting. In a language with precise reference counts, we do not need a distinguished value for holes, but copy contexts eagerly whenever they are shared. The tests (A) and (B) become:

```
// copy c1 ?
struct ctx_t  d1 = (!is_unique(c1.root) ?  ctx_copy(c1) : c1 );                    // (A)

// copy c2 ? (needed to maintain context paths where each node beside the root is unique)
struct ctx_t  d2 = (!is_unique(c2.root) ?  ctx_copy(c2) : c2 );                    // (B)
```

This is the implementation that is used in the Koka runtime system. The (B) check here is required to maintain the invariant that context paths always form *unique chains* [Leijen and Lorenzen 2023]. From this property it follows directly that no cycles can occur in the context path.

C FORMAL ADDRESSC IMPLEMENTATIONS

This appendix shows the formalized AddressC versions of various published insertion algorithms that we have proven correct with respect to the corresponding functional versions in this paper.

The top-down move-to-root insertion by Stephenson [1980] is already shown in Figure 1 (Section 4). The top-down splay tree insertion by Sleator and Tarjan [1985] is again almost line-by-line equal to the published algorithm. Minor deviations arise from the fact that we split the simultaneous assignment in the `rotate` and `link` functions into several single assignments (like a C programmer would do), that we use two contexts `lctx` and `rctx` instead of the equivalent sentinels `left(null)` and `right(null)`, and that we add extra cases to the `heap_splay_insert_td` function to handle the case where the key is already present in the tree.

The bottom-up splay tree insertion follows the same structure as the published, imperative algorithm. But it is not line-by-line equal as we implement the procedure using pointer reversal. However, this highlights the similarity to zippers and is an equally valid implementation strategy; after all, Sleator and Tarjan [1985] introduce bottom-up splay trees as follows:

> *Splaying, as we have defined it, occurs during a second, bottom-up pass over an access path. Such a pass requires the ability to get from any node on the access path to its parent. To make this possible, we can save the access path as it is traversed (either by storing it in an auxiliary stack or by using "pointer reversal" to encode it in the tree structure), or we can maintain parent pointers for every node in the tree.*

Finally, bottom-up move-to-root and bottom-up zip-tree insertion were not described in pseudocode, but our implementation is idiomatic for the pointer-reversal approach.

C.1 Bottom-Up Move-To-Root Tree Insertion

The formalized bottom-up move-to-root algorithm as described by Allen and Munro [1978].

```
Notation "e '->tag'"   := (Load (e%E +ₗ #0%nat)) (at level 20) : expr_scope.
Notation "e '->left'"  := (Load (e%E +ₗ #1%nat)) (at level 20) : expr_scope.
Notation "e '->key'"   := (Load (e%E +ₗ #2%nat)) (at level 20) : expr_scope.
Notation "e '->right'" := (Load (e%E +ₗ #3%nat)) (at level 20) : expr_scope.

Definition rotate_right :=
  fun: ( t ) {
    var: l := t->left in
    var: lr := l->right in
    t->left = lr;;
    l->right = t;;
    t = l
  }.

Definition rotate_left :=
  fun: ( t ) {
    var: r := t->right in
    var: rl := r->left in
    t->right = rl;;
    r->left = t;;
    t = r
  }.
```

```
Definition heap_mtr_rebuild :=
  fun: ( zipper' , tree' ) {
    while: ( true ) {
      if: ( zipper' == #0 ) { break } else {
        if: ( zipper'->tag == #1 ) {
          var: up := zipper'->left in
          zipper'->left = tree';;
          rotate_right (&zipper');;
          zipper' = up
        } else {
          var: up := zipper'->right in
          zipper'->tag = #1;; (* set tag from NodeR to Node *)
          zipper'->right = tree';;
          rotate_left (&zipper');;
          zipper' = up
    } } };; ret: tree'
  }.

Definition heap_mtr_insert_bu :=
  fun: ( i, tree' ) {
    var: zipper' := #0 in
    while: ( true ) {
      if: ( tree' == #0 ) {
        tree' = AllocN #4 #0;;
        tree'->tag = #1;;
        tree'->key = i;;
        break
      } else {

        if: ( i == tree'->key) {
          break
        } else {
          var: tmp := #0 in
          (if: ( i < tree'->key) {
            tmp = tree'->left;;
            tree'->left = zipper'
          } else {
            tmp = tree'->right;;
            tree'->tag = #2;;
            tree'->right = zipper'
          });;
          zipper' = tree';;
          tree' = tmp
    } } };;

    ret: ( heap_mtr_rebuild (&zipper') (&tree') )
  }.
```

C.2 Bottom-Up Splay Insertion

The bottom-up splay tree insertion as shown by Sleator and Tarjan [1985] (Section 4, page 666).

```
Notation "e '->tag'"   := (Load (e%E +ₗ #0%nat)) (at level 20) : expr_scope.
Notation "e '->left'"  := (Load (e%E +ₗ #1%nat)) (at level 20) : expr_scope.
Notation "e '->key'"   := (Load (e%E +ₗ #2%nat)) (at level 20) : expr_scope.
Notation "e '->right'" := (Load (e%E +ₗ #3%nat)) (at level 20) : expr_scope.

Definition rotate_right : val :=
  fun: ( z, t ) {
    var: tmp := z->left in
    z->tag = #1;;
    z->left = t->right;;
    t->right = z;;
    z = tmp
  }.

Definition rotate_left : val :=
  fun: (z, t) {
    var: tmp := z->right in
    z->tag = #1;;
    z->right = t->left;;
    t->left = z;;
    z = tmp
  }.
```

```
Definition heap_splay_rebuild : val :=
  fun: (px, x) {
    while: ( true ) {
      if: (px == #0) {
        break
      } else {
        if: (px->tag == #1) {
          var: gx := px->left in
          if: (gx == #0) {
            rotate_right (&px) (&x)
          } else {
            if: (gx->tag == #1) {
              rotate_right (&gx) (&px);;
              rotate_right (&px) (&x);;
              px = gx
            } else {
              rotate_right (&px) (&x);;
              rotate_left (&px) (&x)
            }
          }

        } else {
          var: gx := px->right in
          if: (gx == #0) {
            rotate_left (&px) (&x)
          } else {
            if: (gx->tag == #1) {
              rotate_left (&px) (&x);;
              rotate_right (&px) (&x)
            } else {
              rotate_left (&gx) (&px);;
              rotate_left (&px) (&x);;
              px = gx
            }
          }
        }
      }
    };;
    ret: x
  }.

Definition heap_splay_insert_bu : val :=
  fun: ( i,  tree' ) {
    var: zipper' := #0 in
    while: ( true ) {
      if: ( tree' == #0 ) {
        tree' = (AllocN #4 #0);;
        tree'->tag = #1;;
        tree'->key = i;;
        break
      } else {
        if: ( i == tree'->key ) {
          break
        } else {
          var: tmp := #0 in
          (if: ( i < tree'->key ) {
            tmp = tree'->left;;
            tree'->left = zipper'
          } else {
            tmp = tree'->right;;
            tree'->tag = #2;;
            tree'->right = zipper'
          });;
          zipper' = tree';;
          tree' = tmp
        }
      }
    };;
    ret: heap_splay_rebuild (&zipper') (&tree')
  }.
```

C.3 Top-Down Splay Insertion

The top-down splay tree insertion as shown by Sleator and Tarjan [1985] (Section 4, page 669).

```
Notation "e '->left'"  := (Load (e%E +ₗ #0%nat)) (at level 20) : expr_scope.
Notation "e '->key'"   := (Load (e%E +ₗ #1%nat)) (at level 20) : expr_scope.
Notation "e '->right'" := (Load (e%E +ₗ #2%nat)) (at level 20) : expr_scope.

Definition rotate_right : val :=
  fun: ( tree' ) {
    var: l := tree'->left in
    tree'->left = l->right;;
    l->right = tree';;
    tree' = l
  }.

Definition rotate_left : val :=
  fun: ( tree' ) {
    var: r := tree'->right in
    tree'->right = r->left;;
    r->left = tree';;
    tree' = r
  }.

Definition link_left : val :=
  fun: ( tree', lhole ) {
    *lhole = tree';;
    lhole = &(tree'->right);;
    tree' = tree'->right
  }.

Definition link_right : val :=
  fun: ( tree', rhole ) {
    *rhole = tree';;
    rhole = &(tree'->left);;
    tree' = tree'->left
  }.

Definition assemble : val :=
  fun: ( tree', lhole, rhole, lctx, rctx ) {
    *lhole = tree'->left;;
    *rhole = tree'->right;;
    tree'->left = lctx;;
    tree'->right = rctx
  }.

Definition heap_splay_insert_td : val :=
  fun: ( i, tree' ) {
    var: lctx := #0 in
    var: rctx := #0 in
    var: lhole := &lctx in
    var: rhole := &rctx in
    while: ( true ) {
      if: (tree' != #0) {
        if: ( i == tree'->key) {
          break
        } else {
          if: ( i < tree'->key) {
            if: (tree'->left != #0) {
              if: ( i == tree'->left->key) {
                link_right (&tree') (&rhole);;
                break
              } else {
                if: ( i < tree'->left->key) {
                  rotate_right (&tree');;
                  link_right (&tree') (&rhole)
                } else {
                  link_right (&tree') (&rhole);;
                  link_left (&tree') (&lhole)
                }
              }
            } else {
              var: l := tree'->left in
              l = AllocN #3 #0;;
              l->key = i;;
              l->right = tree';;
              tree' = l;;
              break
            }
```

```
          } else {
            if: (tree'->right != #0) {
              if: ( i == tree'->right->key) {
                link_left (&tree') (&lhole);;
                break
              } else {
                if: ( i > tree'->right->key) {
                  rotate_left (&tree');;
                  link_left (&tree') (&lhole)
                } else {
                  link_left (&tree') (&lhole);;
                  link_right (&tree') (&rhole)
                }
              }

            } else {
              var: r := tree'->right in
              r = AllocN #3 #0;;
              r->left = tree';;
              r->key = i;;
              tree' = r;;
              break
            }
          }
        }
      } else {
        tree' = AllocN #3 #0;;
        tree'->key = i;;
        break
      }
    };;
    assemble (&tree') (&lhole) (&rhole) (&lctx) (&rctx);;
    ret: tree'
  }.
```

C.4 Derived Top Down Zip Tree Insertion

This is our new top-down zip tree insertion as derived from the functional top-down algorithm in Section 6.2 and 6.4.

```
Notation "e '->rank'"  := (Load (e%E +ₗ #0%nat)) (at level 20) : expr_scope.
Notation "e '->left'"  := (Load (e%E +ₗ #1%nat)) (at level 20) : expr_scope.
Notation "e '->key'"   := (Load (e%E +ₗ #2%nat)) (at level 20) : expr_scope.
Notation "e '->right'" := (Load (e%E +ₗ #3%nat)) (at level 20) : expr_scope.

Definition heap_is_higher_rank : val :=
  rec: "is_higher_rank" "rk1" "rk2" "x1" "x2" :=
    ("rk2" < "rk1") || (("rk1" == "rk2") && ("x1" < "x2")).

Definition heap_unzip_td : val :=
  fun: (x, key, cur) {
    var: accl := &(x->left) in    (* ctx _ *)
    var: accr := &(x->right) in
    while: (cur != #0) {
      if: (cur->key < key) {
        *accl = cur;;              (* accl ++ ctx ... Node(rnk,l,x,_) *)
        repeat: { accl = &(cur->right);; cur = cur->right }
        until: ((cur == #0) || (cur->key >= key))
      } else {
        *accr = cur;;
        repeat: { accr = &(cur->left);; cur = cur->left }
        until: ((cur == #0) || (cur->key < key))
      }
    };;
    *accl = #0;;                   (* accl ++. Leaf *)
    *accr = #0
  }.
```

```
Definition heap_zip_insert_td : val :=
  fun: ( root, rank, key ) {
    var: cur := root in
    var: prev := &root in
    while: ( (cur != #0) && heap_is_higher_rank (cur->rank) rank (cur->key) key ) {
      if: ( cur->key < key ) { prev = &(cur->right);; cur = cur->right }
      else {                   prev = &(cur->left) ;; cur = cur->left  }
    };;
    if: ( (cur != #0) && (cur->key == key) ) {
      ret: root
    } else {
      var: x := AllocN #4 cur in
      x->rank = rank;;
      x->key = key;;
      *prev = x;;
      heap_unzip_td (&x) (&key) (&cur);;
      ret: root
    }
  }.
```

C.5 Derived Bottom-Up Zip Tree Insertion

This is bottom-up zip tree insertion as derived from the functional bottom-up algorithm in Section 6.3 and 6.4.

```
Notation "e '->tag'"   := (Load (e%E +ₗ #0%nat)) (at level 20) : expr_scope.
Notation "e '->rank'"  := (Load (e%E +ₗ #1%nat)) (at level 20) : expr_scope.
Notation "e '->left'"  := (Load (e%E +ₗ #2%nat)) (at level 20) : expr_scope.
Notation "e '->key'"   := (Load (e%E +ₗ #3%nat)) (at level 20) : expr_scope.
Notation "e '->right'" := (Load (e%E +ₗ #4%nat)) (at level 20) : expr_scope.

Definition heap_is_higher_rank : val :=
  rec: "is_higher_rank" "rk1" "rk2" "x1" "x2" :=
    ("rk2" < "rk1") || (("rk1" == "rk2") && ("x1" < "x2")).

Definition heap_rebuild : val :=
  fun: ( zipper', tree' ) {
    while: ( true ) {
      if: ( zipper' == #0 ) {
        break
      } else {
        if: (zipper'->tag == #1) {
          var: tmp := zipper'->left in
          zipper'->left = tree';;
          tree' = zipper';;
          zipper' = tmp
        } else {
          var: tmp := zipper'->right in
          zipper'->tag = #1;;
          zipper'->right = tree';;
          tree' = zipper';;
          zipper' = tmp
        }
      }
    };;
    ret: tree'
  }.
```

```
Definition heap_unzip_bu : val :=
  fun: ( tree', k ) {
    var: zs := #0 in
    var: zb := #0 in
    while: ( true ) {
      if: (tree' == #0) {
        break
      } else {
        if: (tree'->key < k) {
          var: tmp := tree'->right in
          tree'->tag = #2;;
          tree'->right = zs;;
          zs = tree';;
          tree' = tmp
        } else {
          var: tmp := tree'->left in
          tree'->left = zb;;
          zb = tree';;
          tree' = tmp
        }
      }
    };;
    ret: Pair (heap_rebuild (&zs) (ref #0)) (heap_rebuild (&zb) (ref #0))
  }.

Definition heap_zip_insert_bu : val :=
  fun: ( tree', rank, k, acc ) {
    while: ( true ) {
      if: (tree' == #0) {
        tree' = AllocN #5 #0;;
        tree'->tag = #1;;
        tree'->rank = rank;;
        tree'->key = k;;
        break
      } else {
        if: ( heap_is_higher_rank (tree'->rank) rank (tree'->key) k ) {
          if: (tree'->key < k) {
            var: tmp := tree'->right in
            tree'->tag = #2;;
            tree'->right = acc;;
            acc = tree';;
            tree' = tmp
          } else {
            var: tmp := tree'->left in
            tree'->left = acc;;
            acc = tree';;
            tree' = tmp
          }

        } else {
          if: (tree'->key == k) {
            break
          } else {
            var: tmp := heap_unzip_bu (ref tree') (ref k) in
            tree' = AllocN #5 #0;;
            tree'->tag = #1;;
            tree'->rank = rank;;
            tree'->left = Fst tmp;;
            tree'->key = k;;
            tree'->right = Snd tmp;;
            break
          }
        }
      }
    };;
    ret: heap_rebuild (&acc) (&tree')
  }.
```

D EXAMPLE PROOFS

D.1 Move-To-Root Top-Down is Correct

As an example of a typical proof in AddressC with Iris, here is the proof of Theorem 3 that Stephenson's top-down move-to-root insertions is correct (Theorem 3 in Section 4). The main difficulty of this proof lies in specifying the loop invariants `"H"` for the while-loop. The first formula passed to the `wp_while_true` tactic gives the condition once the loop terminates and the second formula gives the invariant for subsequent iterations. The first formula mirrors the *return value* of the functional code (Section 3.3), which returns two trees `l` and `r`. Additionally, it specifies that `left_dummy` and `right_dummy` point to those trees, that `root` points to a Node allocation and that our final result `Node l x r` is equal to the result of the functional code `mtr_insert_td i t`.

The invariant for subsequent iterations mirrors the *recursive calls* of the functional code, which calls itself in tail-position on a tree `t'` and two contexts `lz', rz'`. Additionally, the invariant specifies that `left_dummy` and `right_dummy` point to the contexts and `left_hook` and `right_hook` point to the holes, while `node` points to the subtree and `name` stays constant. Finally, it asserts ownership over the `root` location and asserts that the functional values `lz', rz', t` correspond to a loop iteration of the functional code `mtr_insert_td`.

Once this invariant is given, the proof is mostly automated and it is only necessary to make the case-splits into branches and specify the relevant invocations of the context composition and application lemmas.

```
Lemma heap_mtr_insert_td_correct (i : Z) (tv : val) (t : tree) :
    {{{ is_tree t tv }}}
    heap_mtr_insert_td (ref #i) (ref tv)
    {{{ v, RET v; is_tree (mtr_insert_td i t) v }}}.
Proof.
  wp_begin "Ht"; name, root. wp_var left_dummy. wp_var right_dummy.
  wp_load. wp_var node. wp_var left_hook. wp_var right_hook. wp_while_true "H"
    (∃ l (x : Z) r lv rv (p : loc) lv' rv',            (* condition on exiting the loop *)
            root ↦ #p ∗ p ↦∗ [lv'; #x; rv']
            ∗ left_dummy ↦ lv ∗ is_tree l lv
            ∗ right_dummy ↦ rv ∗ is_tree r rv
            ∗ ⌜mtr_insert_td i t = Node l x r⌝)%I
    (∃ lz' rz' t' (lhv rhv : loc) lhvv rhvv rootv treev, (* invariant during iteration *)
            node ↦ treev ∗ is_tree t' treev
            ∗ root ↦ rootv ∗ name ↦ #i
            ∗ left_hook ↦ #lhv ∗ lhv ↦ lhvv ∗ is_ctx lz' left_dummy lhv
            ∗ right_hook ↦ #rhv∗ rhv ↦ rhvv ∗ is_ctx rz' right_dummy rhv
            ∗ ⌜mtr_insert_td i t = mtr_insert_td_go i lz' rz' t'⌝)%I.
  - iDecompose "H". wp_heap. wp_type.
  - iDestruct "H" as (lz' rz' t' lhv rhv lhvv rhvv rootv treev) "[? [Ht H]]".
    iDecompose "H". rewrite H. unfold td_insert_go at 1.
    destruct t' as [|l x r].
    + iDestruct "Ht" as %->. wp_heap.
      iPoseProof (tree_of_ctx lz' Leaf left_dummy lhv) as "?".
      iPoseProof (tree_of_ctx rz' Leaf right_dummy rhv) as "?".
      wp_type.
    + iDestruct "Ht" as (p l' r') "[-> [? [? ?]]]". wp_heap.
      case_bool_decide; wp_heap.
      { iPoseProof (tree_of_ctx lz' l left_dummy lhv) as "?".
        iPoseProof (tree_of_ctx rz' r right_dummy rhv) as "?".
        wp_type. }
      { case_bool_decide; wp_heap.
        { iPoseProof (ctx_of_ctx rz' (Node0' x r) right_dummy rhv) as "?".
          wp_continue lz', (comp rz' (Node0' x r)), l. }
        { iPoseProof (ctx_of_ctx lz' (Node2' l x) left_dummy lhv) as "?".
          wp_continue (comp lz' (Node2' l x)), rz', r. } }
  - wp_type.
Qed.
```