

Benchmarking a Baseline Fully-in-Place Functional Language Compiler

Jaromír Procházka,
Vít Šefl^[0000–0003–4350–9256], and Tomas Petricek^[0000–0002–7242–2208]

Charles University, Czech Republic
jaromir.prochazka724@student.cuni.cz
sefl@ksvi.mff.cuni.cz
petricek@d3s.mff.cuni.cz

Abstract. Functional programming makes code easier to reason about, inherently thread-safe, and more testable. However, immutable data structures introduce efficiency costs, as many operations require copying rather than performing in-place modifications. To address this, the Koka language introduced a novel mechanism called fully-in-place functional programming (FIP). The mechanism enables safe in-place updates while minimizing unnecessary memory allocations. Nevertheless, Koka’s garbage collection and extensive feature set complicate the task of isolating FIP’s specific memory efficiency advantages. We design a minimal functional language that supports fully in-place updates based on the FIP calculus while omitting garbage collection. This lets us compare the performance of the FIP approach with that of a conventional implementation. Using finger trees, red-black trees, and quicksort as case studies, we show that a language employing the FIP calculus in a garbage-collection-free environment can achieve a significant increase in performance and memory efficiency. Our work confirms the results of the original authors, but it also uncovers potential limitations of the FIP approach.

Keywords: Fully-in-place functional programming · Compilation

1 Introduction

Programs written in functional languages achieve their advantages through immutability, but it is no secret that functional data structures allocate a lot of memory. Updating these structures is done through copying rather than in-place mutations, and even though parts of the structure can typically be shared, some additional memory needs to be allocated. Fortunately, most allocations are ephemeral, and thus the residency of functional programs (the total amount of memory in use at any given point) is typically not large.

Modern generational garbage collectors are good at handling the memory allocation patterns of functional programs [9]. In particular, garbage collection performs best for short-lived structures that never leave the first generation of GC. However, even though the cost of allocation (and deallocation) has gotten very low, it is not free.

In some cases, memory allocation can be replaced with in-place mutation while still maintaining functional purity. That is, destructive updates can occur as long as they are not observable. For example, if we copy an array and the original immediately ceases to exist (there are no observers that can access it), the copying can be elided and the update done in-place. However, identifying these optimization opportunities automatically can be quite challenging.

The FIP (*fully in-place*) calculus [6] is a recent attempt at solving this problem in a principled way. The FIP calculus describes a wide variety of purely functional programs that can be executed safely using in-place updates, thus avoiding unnecessary memory allocation. If a function argument is *unique* (i.e., not aliased) and *owned*, the memory it occupies can be reused as it is deconstructed. FIP itself does not specify how to ensure that the reused structure is unique and leaves it up to the implementation. Uniqueness could be tracked statically (e.g. through linear types) or dynamically (e.g. through reference counting).

The FIP calculus has been successfully implemented in the Koka language [5]. Although Koka shows promising results, to the best of our knowledge, it is a singular experiment. Koka is a rich language with a more complex reference-counting runtime, which complicates the evaluation of benefits provided by FIP. To validate the promising results of Koka, we aim to examine whether it is possible to obtain the same benefits in an independent functional language.

Goals and Contributions The primary objective of this work is to design and implement a small, compiled functional language based on the FIP calculus. The implementation has a minimalistic runtime. In particular, no garbage collection or reference counting should be employed.

With the implementation in place, the next step is to reproduce the experiments of the original paper [6]: quicksort, red-black tree insertion, and finger tree insertion. We are interested in the relative speedup between a standard re-allocating implementation and a FIP implementation. Finally, we analyze the results and compare them with those of the original paper.

As a secondary objective, we analyze how the design of the FIP-based type system interacts with the chosen compiled data representation. This points to potential limitations of the FIP approach. Specifically, our contributions are:

- We implement a small functional language StaFip (Section 3). Unlike Koka, StaFip is a statically compiled language without garbage collection. Its features were chosen to provide a strong baseline for benchmarking FIP.
- We reimplement three benchmarks from the original paper [6] (finger trees, red-black trees, quicksort) using StaFip and compare the performance of a conventional and FIP-based compilation strategy. The speedup (1.4–3x) closely resembles the speedup measured in the context of Koka, indicating the validity of the FIP approach (Section 4).
- We observe that the type system can be sensitive to implementation details (Section 5). In particular, a valid FIP algorithm that works in one language might not work in another language. The degree to which validity depends on implementation details could pose an issue to FIP.

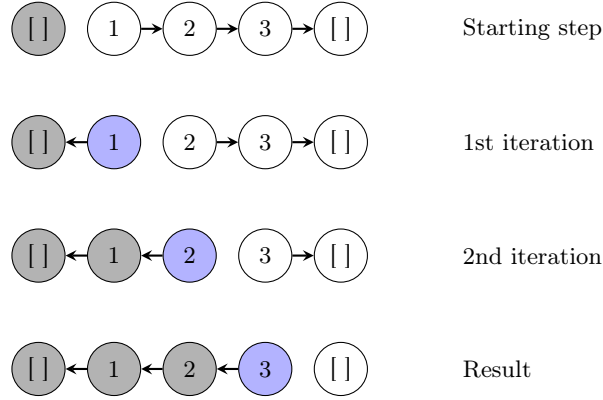


Fig. 1: Step-by-step execution of the fully in-place list reversal.

2 Background

2.1 The FIP Approach

To introduce the FIP approach, we consider an example presented in the original paper. In a standard functional implementation of a list reversal, the program recursively traverses the list using an accumulator parameter. For each item of the list, a new node is allocated and added to the accumulator, as can be seen in Figure 2.

The FIP approach avoids this inefficiency by reusing the space of the (decomposed) **reversed** list to initialize the (**_head:accumulator**) node without the need for a new allocation. It is important to keep in mind that the **reversed** argument must be unique to the FIP function to maintain its functional purity (no observable side effects). To understand how the new node can be initialized without allocation, we need to look at *reuse tokens*, which is a mechanism that the FIP calculus uses to keep track of available memory. When decomposing the **reversed** list, the cons cell (**_head:rest**) is stored in a block of memory with two elements, the head and the tail. When the cons cell is deconstructed, the memory is available for reuse with a reuse credit of size 2 (written as $\diamond_2$). When reusing the memory during the allocation of (**_head:accumulator**), the FIP calculus ensures that the size of the new structure fits the size of the available reuse credit (as seen in Figure 1).

These reuse tokens must represent a contiguous segment of memory. Note that two reuse tokens representing adjacent segments cannot be merged into a larger reuse token. This is because we cannot *statically* ensure that these two segments are allocated right next to each other.

```

reverse :: [a] -> [a] -> [a]
reverse reversed accumulator =
  case reversed of
    -- (_head : accumulator) allocates new list node
    (_head:rest)  -> reverse rest (_head : accumulator)
    []           -> accumulator

main :: IO ()
main = print $ reverse [1, 2, 3] []

```

Fig. 2: A Haskell code representing a simple allocating list reversal.

2.2 The FIP Calculus

The original FIP calculus models a fragment of a functional language embedded in a larger language. For benchmarking purposes, we only need the smaller fragment and so we omit aspects that exist only thanks to the larger language such as general function application. We keep application of first-order functions, pattern matching on expressions, and reuse of available data blocks [6, p. 9].

Expressions, values and patterns:

$E ::= (V, \dots, V)$	(tuple)
$\mid f(E; E)$	(call)
$\mid \text{match } E \{P \mapsto E\}$	(match)
$\mid \text{match! } E \{P \mapsto E\}$	(destructive match)
$V ::= x$	(variable)
$\mid c \ V \ V \ .. \ V$	(constructor)
$P ::= C \ x_1 \ \dots \ x_k$	(pattern)

The syntax of the FIP calculus distinguishes between values and expressions. Expressions include tuple construction, function call, match and destructive match. The match terms are used for branching and unboxing of data in the program. The destructive match (**match!**), in particular, is a special FIP construct that destructs the argument and allows the reuse of its memory block. Values are variables and constructors applied to other values.

For reusing memory, we denote a reuse token of size k as $\diamond_k$. These tokens represent contiguous parts of allocated memory available for reuse. They are created from data destroyed through the **match!** expression.

As mentioned above, these blocks cannot be merged or split. Merging would not always preserve their contiguity or would lead to reallocations, which is exactly what we are trying to avoid. Splitting, on the other hand, could create reuse tokens of size unusable by any constructor. It would also make it harder to manage memory. A memory block that has been split can only be deallocated

$$\begin{array}{c}
\frac{}{\Delta \mid x \vdash x} \text{VAR} \qquad \frac{\Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma, \diamond_0 \vdash e} \text{EMPTY} \qquad \frac{}{\Delta \mid \emptyset \vdash C} \text{ATOM} \\
\\
\frac{}{\Vdash \emptyset} \text{DEFBASE} \qquad \frac{\Vdash \Sigma' \quad \bar{y} \mid \bar{x} \vdash e}{\Vdash \Sigma', f(\bar{y}; \bar{x}) = e} \text{DEFFUN}
\end{array}$$

Fig. 3: Well-formedness rules of the FIP calculus (variables and base terms)

once none of its parts are in use, leading to memory fragmentation and increased overhead from the required tracking.

Since the FIP approach reuses allocated memory blocks by destroying them and converting them to reuse tokens, we need to ensure that this data is not used by other parts of the program. For that purpose, the FIP calculus uses two separate variable contexts:

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

Δ denotes the borrowed environment, with values that are not owned by us, and thus must not be destroyed. Γ denotes the owned environment, with data not shared with other parts of the program. This data can be safely reused, including the reuse tokens themselves. The judgment $\Delta \mid \Gamma \vdash e$ then means that the expression e is a well-formed FIP expression in the borrowed context Δ and the owned context Γ . A union of two contexts is denoted by Δ, Δ' .

The FIP calculus itself is type agnostic. It only captures the essence of a FIP functional language fragment. Some interesting scenarios, such as static tracking of ownership, could rely on the type system used in the target language.

Well-formedness. A FIP term is well-formed if every allocation reuses memory of previously destroyed data. This is ensured using the well-formedness judgment written as $\Delta \mid \Gamma \vdash e$. Figure 3 shows the base and variable rules.

As previously mentioned, these rules are independent of any particular type system. It is also important to note that realistic programs cannot be expressed with only the FIP calculus. There must be some way of integrating FIP code fragments in a larger non-FIP program. Indeed, memory reuse is only possible after a prior memory allocation, which the FIP calculus cannot express.

The VAR rule states that any variable in a borrowed environment is well-formed. The baseline rules (DEFBASE, ATOM, EMPTY, DEFFUN) handle the trivial cases of definitions, constructions, and reuses. Here, the $\Vdash \Sigma'$ notation means that the set of top-level functions (the signature Σ') is fully in-place. The DEFFUN rule shows that a function definition is well-formed if its body is well-formed in the contexts of the parameters.

The remaining interesting rules are shown in Figure 4. The TUPLE rule states that we can create tuples of values which are well-formed in the borrowed context and the union of the owned contexts of the values that make up the tuple.

$$\begin{array}{c}
\frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \dots, \Gamma_n \vdash (v_1, \dots, v_n)} \text{ TUPLE} \qquad \frac{\bar{y} \in \Delta, \text{dom}(\Sigma) \quad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash f(\bar{y}; e)} \text{ CALL} \\
\\
\frac{y \in \Delta \quad \Delta, \bar{x}_i \mid \Gamma \vdash e_i \quad \bar{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma \vdash \text{match } y \{C_i \bar{x}_i \mapsto e_i\}} \text{ BMATCH} \\
\\
\frac{\Delta \mid \Gamma, \bar{x}_i, \diamond_k \vdash e_i \quad k = |\bar{x}_i| \quad \bar{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma, x \vdash \text{match! } x \{C_i \bar{x}_i \mapsto e_i\}} \text{ DMATCH!} \\
\\
\frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \dots, \Gamma_k, \diamond_k \vdash C^k v_1 \dots v_k} \text{ REUSE}
\end{array}$$

Fig. 4: Well-formedness rules for tuples, calls, pattern matching and reuse

The rules BMATCH and DMATCH! are used to create branching and unboxing in the program. The non-destructive BMATCH is well-formed if the matched object y is in a borrowed context, each expression e_i matched with the pattern $C_i \bar{x}_i$ is well-formed, and the pattern variables $\bar{x}_i$ are not already in the borrowed or owned contexts. They are new, unique variables (in a practical implementation, shadowing does not impact the FIP mechanism). The destructive DMATCH! rule states that the expression is well-formed if each expression e_i following the pattern is well-formed in the context with $\diamond_k$ reuse token included, provided by the matched expression x . Notice that this is the only rule where the reuse token is introduced to a context. It provides these reuse tokens for the REUSE rule. This rule states that a construction of an object C^k using a reuse token of the same size $\diamond_k$ is well-formed if its arguments are well-formed.

CALL rule states that, with some number of arguments in borrowed context or bounded in the global environment ($\text{dom}(\Sigma)$) and a well-formed resulting expression, a function call with these arguments as parameters is well-formed.

2.3 Koka Implementation

Koka is a strongly typed functional-style research language with effect types and handlers [5]. It implements the FIP calculus using the approach outlined above, as an embedded language.

Koka uses reference counting to manage its memory and to ensure the uniqueness of values to be reused. If a function argument has a reference count of one, then we can be sure that its memory is not accessed anywhere else in the program and thus can be safely reused. In the formal language of the FIP calculus, all the arguments of a FIP function with the reference count of one are in the owned context of that function call.

Koka also provides an option of compiling directly to C code using the Perceus method [8] for reference counting. This method uses some aggressive optimiza-

```
int main [int argc, char** argv] =
    printf("Hello world!")
```

Fig. 5: Example of main function definition.

tions and static analysis in order to generate code without the need for a garbage collector or runtime system.

Furthermore, Koka implements an extension of FIP called FBIP (Functional But In-Place). It allows FIP to also deallocate when necessary. In theory, this is achieved by adding store semantics to the FIP syntax with new *drop* and *free* operations. *drop x* is used for dropping an owned variable *x* from the owned environment and the *free k* is used for freeing a reuse token of size *k* [6, p. 12].

The garbage collection with reference counting creates an overhead not present in the unique typed environments. This approach is also not efficient for contexts with small memory, like microcontrollers. Although Koka tries to bridge this gap with the aforementioned Perceus method, this requires aggressive optimizations.

3 StaFip Implementation

To test the FIP approach, we designed a lightweight functional language called StaFip [7]. StaFip is a statically compiled functional language. Unlike Koka, it does not use reference counting, but manual memory management. It is made to provide a strong baseline for benchmarking the FIP approach as described in the background, without unrelated features from Koka possibly confounding the testing results.

Let us now discuss how the StaFip language represents data types and how the FIP rules are enforced.

3.1 Basic StaFip Terms

A StaFip program consists of algebraic type definitions and function definitions, one of which must be named `main` (Figure 5). Type definitions start with a `type` keyword and specify a list of constructors as can be seen in Figure 6.

Function declaration can either include a body or simply declare a function without it. This allows us to declare a function and provide its definition later. A function declaration consists of an optional `fip` keyword for FIP functions, a return type, a name, a list of explicitly typed parameters, and optionally a body.

Inside the body, we can nest a set of simple expressions such as constructors, function calls, or arithmetic operations, as well as, complex expressions like match, destructive match and if-else expressions.

3.2 Compilation of Types

StaFip supports custom data types defined in a similar fashion to Haskell. An example in Figure 6 shows a list data type definition, which can be either a `Nil`

```

type list {
    Nil;
    Cons(int val, type list next);
}

// Instantiation: Cons( 1, Cons (2, Nil ) )
// results in list: 1 -> 2 -> Nil

```

Fig.6: Example of list enum type definition

or a `Cons` with a value and list data fields inside it. The StaFip implementation represents these data types in a way similar to C structs. When using the type as an argument, for instance in `type list next`, this type actually translates to `struct list *xs`. The first field of any non-null type is implicitly an `int` tag with a unique type identifier. Another example with a binary tree type is shown in Figure 7 with its memory layout in the diagram below the type definition.

Allocation and Reuse. For each custom data type, an allocating constructor function and a reuser function are implicitly defined.

- A reuser is a low-level function that resembles a constructor function; however, instead of allocating memory on the heap, it takes a pointer to a heap memory block given as its first argument and initializes the data type in that location, effectively reusing the address. This function corresponds to the consumption of the reuse tokens in the formal FIP model.
- The constructor function, by contrast, calls a StaFip standard library memory allocation function, which is a simple wrapper around the C++ `std::malloc` function from `stdlib.h`.

However, `std::malloc` might overestimate the cost of memory management when compared to a modern functional language with a generational garbage collector. In a generational GC, new objects are allocated in a dedicated memory area (a *nursery*). Allocation is simply a matter of incrementing a pointer (*bump pointer allocation*). Once this pointer reaches the end of the nursery, the GC finds all live objects, moves them to the next generation, and resets the nursery. For this reason, we have also implemented a second version of the constructor function, which uses bump pointer allocation. We discuss the relation between these two versions in Section 4.

Data Representation. Data type instances holding some fields are represented as memory allocated blocks similar to C structs with an implicit tag field of type `int`. This tag is a number identifying the type.

This is different for constructor calls with no arguments like the `Nil` constructor in the list example. The resulting type instances are represented as a `NULL` pointer. Consequently, only one such empty constructor can exist in each data

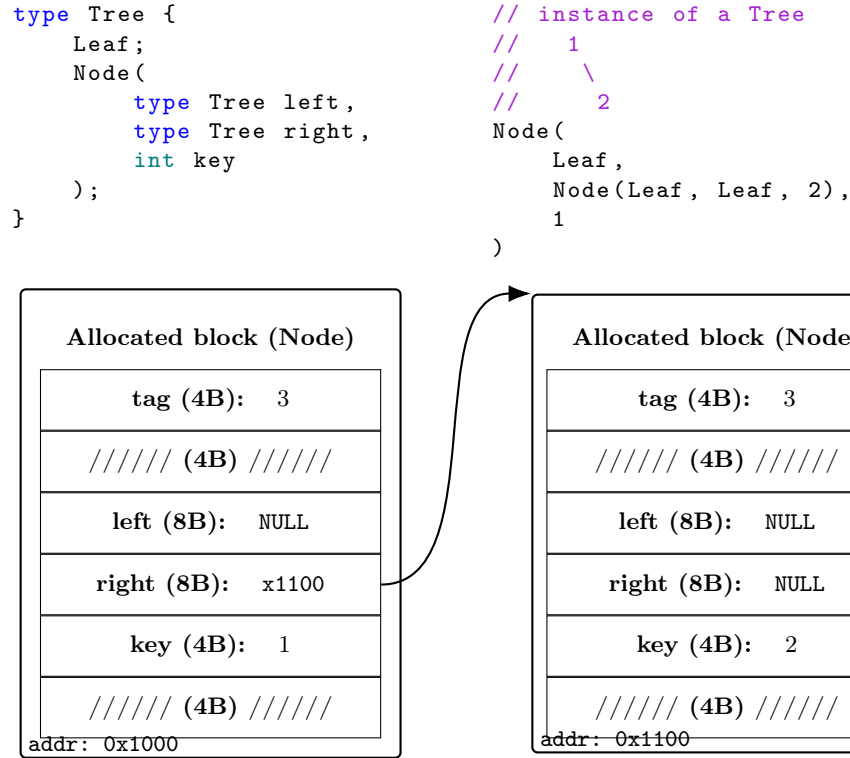


Fig. 7: Example of a binary tree data type, a two-node instance and a diagram showing its heap-allocated layout.

type. In our implementation, multiple empty constructors are indistinguishable from each other at runtime. This is not a problem for our specific benchmarks. Moreover, we could update the StaFip compiler to make the instances of the same constructor with no fields point to some small global memory-allocated block (a different one for each parameter-less constructor). This way, we could distinguish parameter-less constructors at runtime by comparing the pointers to these global memory blocks. However, the problem hints at a potential limitation of the FIP approach that we revisit in Section 5.1.

Binary Tree Example. Let’s look at a more complex example, and its memory layout (Figure 7). This example shows a memory layout for a simple instance of a binary tree. We can see the two data nodes allocated on the heap, with one pointing to the other. Each contains a tag that marks by which constructor this instance was made (Node). The leaf nodes are represented as NULL pointers, and so are not allocated.

```

// list definition from above

fip type list freverse_acc [type list xs, type list acc]
  = match! xs -> type list {
    | Nil          -> acc
    | Cons(x, xx)  -> freverse_acc(xx, Cons(x, acc))
  }

```

Fig. 8: Example of fully in-place list reversal function. Shows a use of destructive match on a list from figure 6.

3.3 Match Expressions

A match expression is where the magic of the FIP method happens. A match expression is a syntactic structure that, given an input expression, creates branching in the program based on the expression’s constructor. In StaFip, we distinguish between two kinds of matches: destructive `match!` and non-destructive `match`. In the context of FIP and the destructive `match!`, it is also where the captured data blocks are reused.

Figure 8 shows a use of the destructive match in a simple list reversal function (also presented in [6]). In this example, we match an expression `xs`. As StaFip has only limited type inference, we also specify type `type list`. If `xs` is a `Nil`, the result of the function is the value of the `acc` variable. If the expression `xs` is `Cons(x, xx)`, the value is deconstructed into the element value `x` and the rest of the list `xx`. In this case, the result of the function is evaluated as the value of a function call `freverse_acc(xx, Cons(x, acc))`. The constructor of the matched expression is checked using the constructor tag allocated on the heap in the case of the `Cons` case, or by checking it to be `NULL` in the case of `Nil`.

But how is the match used for memory block reuse? When destructive `match!` is applied to an expression whose value is allocated on the heap, this value is tracked by the compiler as reusable. While compiling, in each branch of the match expression, if a new data type instance is created, instead of generating an allocating constructor function, a reuser function is generated instead, overwriting the matched expression value in memory. For that reason, it is important to make sure that the caller owns this value. However, since this check would be done statically and thus has no effect on benchmarks, StaFip places this responsibility on the user.

3.4 Implementation of Reuse Tokens

When the compilation process enters a destructive match or body of a FIP function (function with a `fip` keyword in its declaration), the compiler switches into the FIP mode. In this mode, for any constructor call, it will try to find a suitable reusable block of memory and reuse it. Depending on the compiler’s configuration, if there is no such block, it will either report a compilation error or simply

```

1.  match! x1 -> type T1 {           // reusables: empty
2.      | Ta(a, b, c) ->             // reusables: x1{3}
3.          match! x2 -> T1 {       // reusables: x1{3}
4.              | Tb(d, e) ->       // reusables: x1{3}, x2{2}
5.                  Taa(           // reusables: x2{2}
6.                      Tbb(a, b), // reusables: empty
7.                      c, d       // reusables: empty
8.                  )             // reusables: empty
9.              // | ...          // reusables: x1{3}
10.          }                     // reusables: empty
11. // | ...                       // reusables: empty
12. }                              // reusables: empty

```

Fig. 9: Example of nested match expression

allocate instead (weak configuration mostly for debugging). It is important that the FIP mode can be turned off, since in a program with only FIP functions, no memory allocation could ever occur. But how does the compilation process keep track of the reusable blocks?

A StaFip program is compiled in a single traversal. When calling a destructive match on some variable `x1`, this variable (a reference to some heap memory block) is tracked as reusable by a `FipState` singleton defined in the StaFip compiler. Then, if an allocation were to occur, the `FipState` instead provides the reusable memory block of the required size, and a reuser function with this block is generated instead of the allocation.

This process is illustrated in Figure 9. There, we have nested destructive `match!` expressions on argument `x1` and `x2`. The comments besides each line describe at that point of compilation, which arguments are understood by the compiler as reusable.

On the first line of the example 9, we destructively match the argument `x1`. To know what its size is in memory, we need to know its constructor. On the second line, we have a case of `x1` being an instance of `Ta` constructor, which tells us that in this case, `x1` has size 3, since it holds three fields. We thus place $\diamond_3$ to `FipState` and evaluate the case expression on lines 5 to 8. On line 5, we create a constructor with 3 fields, thus we can reuse the `x1` from the `FipState`. On line 6, we are constructing a type instance with 2 fields, so we can reuse the `x2` from the `FipState`. There are no further constructions in this case and we have used all the reusable blocks, so the context is empty. On line 9, we have another case of the inner match expression. We are outside of the context of the last case, and so the `x1` block is still available for reuse. The type of `x2` might be different and so it is not in tracked by `FipState` yet. Outside of the inner match expression, the type of both arguments are unresolved and so the `FipState` is empty.

As shown in the example, the tracking of reusable blocks is tied to variable contexts and scopes. This is best illustrated by the two branches of the pattern matching construct starting on line 3. In the first branch (lines 4–8), the `x1`'s

memory block is reused and so it is not available at the end of the scope. In the second branch, the `x1` memory block reappears on line 9, because the line starts a new scope, associated with a different variable context.

The `FipState` singleton keeps track of the reusable blocks in these contexts at compile time. It holds a stack of contexts each with a set of reusable variables with heap addresses and their sizes.

A new scope is entered, for example, on a match or an if-else expression. When a new scope is entered, and the program is in FIP mode, a new context instance with a set of reusable blocks is appended to the stack. This new context is a copy of the one before it because the new context has access to all reuse tokens from the context above it. When leaving the context block, this instance is again popped from the stack.

4 Benchmarks

For benchmarking purposes, we chose quicksort, red-black tree insertion, and finger tree insertion. Quicksort is a sorting algorithm that serves as a classic benchmark for non-trivial problems. The repeated insertion into a red-black tree was chosen as a test case to illustrate a scenario in which the operation necessitates the allocation of new space, as new data is added. And the finger tree insertion represents a more complex algorithm, closely aligned with practical applications in functional programming.

These algorithms were also benchmarked in the FP² paper [6, p. 25]. They were selected to directly compare the efficiency of the garbage-collected Koka with the statically compiled StaFip. The implementations of these algorithms are designed to minimize bias in the comparisons as much as possible.

These benchmarks (Figure 10) can be run and the plot for the benchmark results generated in the StaFip compiler repository [7]. The resulting plot visualizes these results through two rows of column graphs. The upper row for each algorithm compares the means of the FIP (orange) and reallocating (blue) versions. Above these comparisons, the fetch time is depicted in gray. It describes the portion of the total measured time it takes to prepare the testing data (for the quicksort test, it is the time it takes to first build the unsorted list). Below each comparison, the relative speedup of the FIP implementation is calculated as *reallocating mean / FIP mean*.

In addition to the two columns, the third (purple) column shows a measurement of the same FIP algorithm as the orange column, but with the reusing mechanism disabled (all destructive matches turned to non-destructive matches). It shows the overhead of the FIP algorithms like type paddings, longer code, the use of zippers, and so on, if the reuse mechanism from FIP calculus was not present.

Finally, as discussed above, we have two memory allocation strategies. The `std::malloc` version of the constructor and the bump pointer version. For each of these, we provide a separate measurement.

The `std::malloc` version overestimates the cost of allocation when compared to a state-of-the-art garbage collector. Because of that, it represents the pessimistic scenario of non-FIP code. On the other hand, the bump pointer allocation represents the absolute best case scenario: allocation is essentially free and there is no overhead from nursery management (since the nursery is effectively infinite). These two versions thus give us the upper and lower bounds on the cost of allocation in a realistic GC.

The results of the benchmarks, shown in Figure 10 (standard malloc) and 11 (bump pointer) are:

1. **Finger tree insertions** implemented in FIP are around 1.65 to 2.95x faster than their reallocating implementation in StaFip
2. **Red-Black tree insertions** implemented in FIP are around 1.00 to 1.91x faster than their reallocating implementation in StaFip
3. **Quick sort** implemented in FIP is around 0.81 to 1.35x faster than its reallocating implementation in StaFip

These results, as presented, agree with those in the original paper [6, p. 25]. Although in some cases, like the quicksort algorithm, the FIP approach can introduce slowdowns compared to the non-FIP version with the bump pointer allocator. But even in this worst case situation, other FIP algorithms still offer noticeable performance gains. The FP² paper details their implementation for

1. **Finger trees insertions:** 1.59x faster compared to std,
2. **Red-Black trees insertions:** 2.47x faster compared to std,
3. **Quick sort:** 1.69x faster compared to std

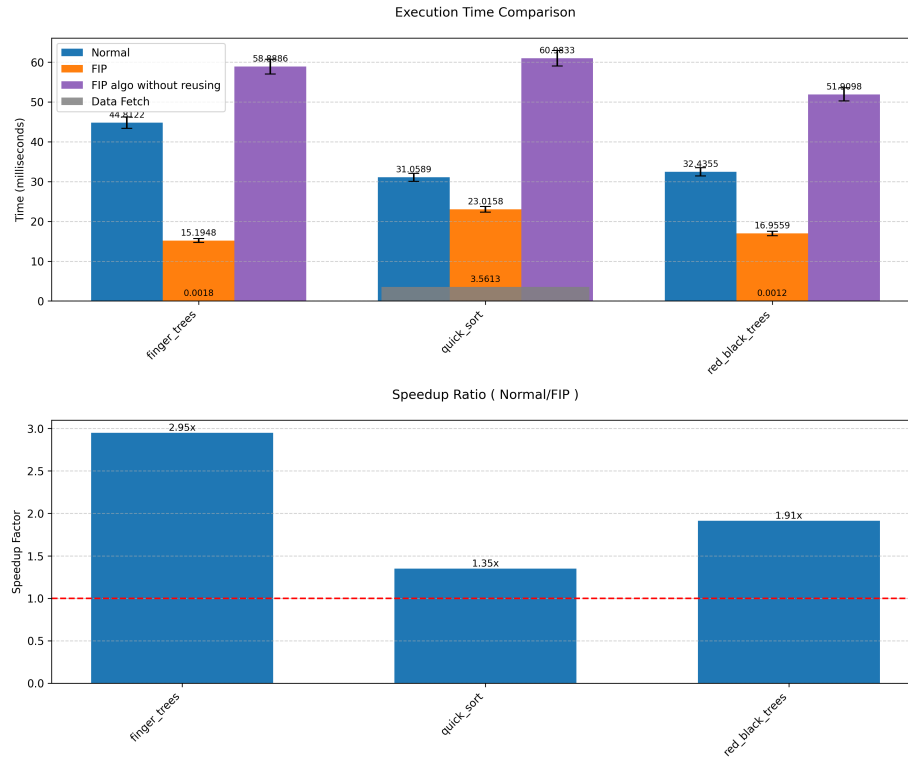
5 Observations

5.1 FIP Implementation

An interesting consequence of the FIP calculus discussed in this paper is its implementation sensitivity. Since type size plays a crucial role in the FIP calculus, algorithms utilizing these types are sensitive to their representation. Consequently, the same algorithm may not be easily translatable into a different language that implements FIP.

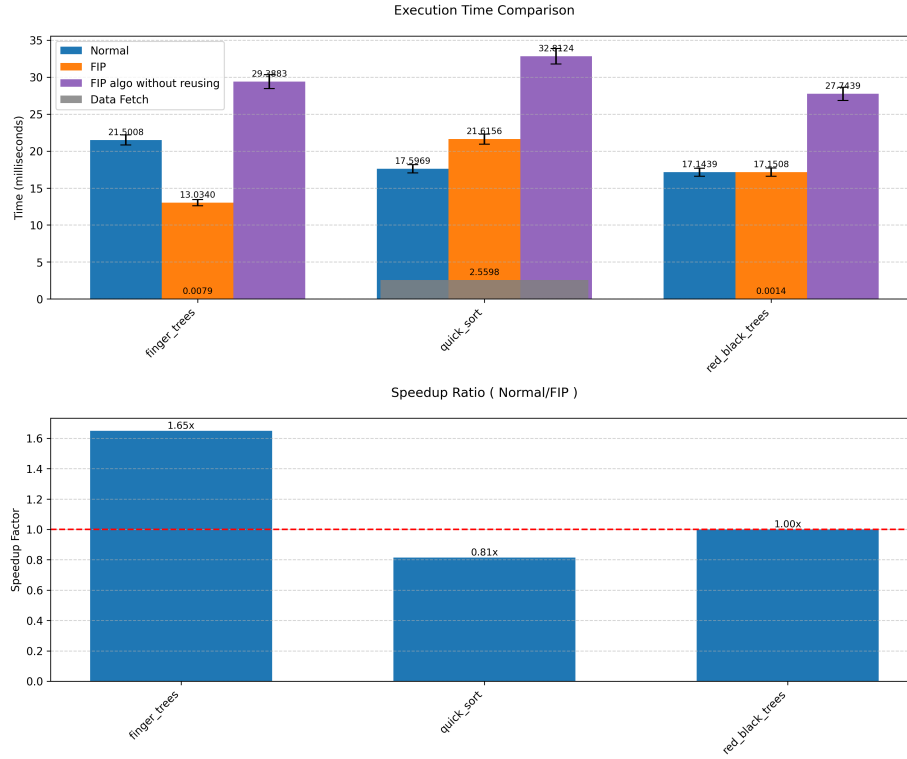
This problem is best illustrated through the Example 12. In this example, we have a **Tree** and a **Buffer** data structure. Both have constructors padded to the size of 3. The **Buffer** type is similar to a list with an extra field for an additional data buffer. We also assume that there is a `concat` method that, given two buffers, concatenates them. The **Tree** type is a binary tree with an additional data buffer field in its leaves. The `linear` method goes through the binary tree using the DFS algorithm and, in that order, generates (in place) a buffer. Elements of this buffer contain a buffer with extra data for the **Leaf** elements in the tree.

Now in StaFip, the `linear` function is a valid fully in-place method. But if the **Empty** constructor was implemented differently in StaFip and was represented by



command	fetch_time	mean	stddev
finger_trees_normal	0.0011	44.8122	1.4183
finger_trees_fip	0.0018	15.1948	0.4811
finger_trees_nivalgo	0.0017	58.8886	1.865
quick_sort_normal	3.5456	31.0589	0.9832
quick_sort_fip	3.5613	23.0158	0.7292
quick_sort_nivalgo	3.5564	60.9833	1.9301
red_black_trees_normal	0.0012	32.4355	1.0281
red_black_trees_fip	0.0012	16.9559	0.5375
red_black_trees_nivalgo	0.0013	51.9098	1.6451

Fig.10: Results of the benchmarks using the `std::malloc` from `stdlib.h` in milliseconds. The "nivalgo" refers to the not-in-place algorithm, which is the algorithm used in the FIP version, but with reusing turned off (all match! replaced by match). It was run on Ubuntu 22.04.4 (AMD Ryzen 5 4600HS). Repeated 1000 times.



command	fetch_time	mean	stddev
finger_trees_normal	0.0011	21.5008	0.6866
finger_trees_fip	0.0079	13.034	0.4134
finger_trees_nipalgo	0.0082	29.3883	0.9369
quick_sort_normal	2.5125	17.5969	0.5595
quick_sort_fip	2.5598	21.6156	0.6853
quick_sort_nipalgo	2.534	32.8124	1.0469
red_black_trees_normal	0.0013	17.1439	0.5485
red_black_trees_fip	0.0014	17.1508	0.5448
red_black_trees_nipalgo	0.0013	27.7439	0.8937

Fig.11: Results of the benchmarks using the **custom bump pointer malloc** function in milliseconds. The "nipalgo" refers to the not-in-place algorithm, which is the algorithm used in the FIP version, but with reusing turned off (all match! replaced by match). It was run on Ubuntu 22.04.4 (AMD Ryzen 5 4600HS). Repeated 1000 times.

```

type Buffer {
  BCons(int key, type Buffer next, type Buffer data);
  Empty;
}

// we also have defined a 'concat' FIP method which takes two
// Buffers and concatenates them as lists

type Tree {
  Node(type ATree l, type ATree r, int i);
  Leaf(int i, type Buffer b, int _padding);
}

fip type Buffer linear [type BTree s] =
  match! s -> type Buffer {
    | Node(l, r, i) ->
      BCons(i, concat(linear(l), linear(r)), Empty)
      // <-- Empty sensitive to implementation
    | Leaf(i, b, _2) ->
      BCons(i, Empty, b)
      // <-- Empty sensitive to implementation
  }

```

Fig. 12: A FIP function which takes a binary tree with extra data (buffers) in its nodes. The function generates a list of the items from the tree (destructively in-place). These list items are in the order of the DFS algorithm’s traversal of the original tree.

some heap-allocated block, this algorithm would no longer be fully in-place, since there would be no reuse token available for allocating the `Empty` value. This is interesting because the FIP calculus does not specify how data types should be implemented. Consequently, two functional languages that implement FIP could have algorithms written in them, where the algorithm is not directly translatable from one language to the other.

5.2 Caching Behavior

Another aspect to consider is how the StaFip programs generally relate to hardware caches. FIP programs limit the amount of reallocations and so are inherently more cache-friendly. A reallocation of an object in the standard approach often leads to a cache miss when this object is accessed again. The cache-friendliness of the FIP approach is also one of the reasons why the programs running using the FIP approach can be much faster than their reallocating counterparts.

This idea is further strengthened by the results of the bump pointer allocator version of the benchmarks, where the cost of allocation is eliminated. As can be seen in Figure 11, even in this situation, the FIP approach can produce a

noticeable speedup, particularly in the case of the finger tree insertion algorithm. However, measuring the precise contribution of caching is beyond the scope of this work.

5.3 StaFip Limitations

The primary concern regarding the testing process is the lack of a deallocation mechanism. The issue is minimal for the implementation of the FIP calculus within the compiler. This is because the FIP algorithm neither allocates memory nor requires deallocation.

Furthermore, this implementation of the StaFip compiler is relatively simple and is not representative of state-of-the-art compilers. However, real-world compilers may introduce distortions to the results. For example, when comparing a FIP algorithm implemented in Koka with a reallocating implementation of the same algorithm in C++, it remains uncertain whether differences between these two languages affect the results. To mitigate this issue, our aim is to maintain consistency in the implementations, with the only variation being the FIP optimization.

5.4 Threats to Validity

There are a number of aspects of the StaFip compiler and the benchmarking process that may threaten the validity of our results. The lack of deallocations may pose a problem for the reallocating algorithm since it may lead to excessive page faults in the operating system. These page faults may occur more frequently because the allocated heap blocks are not freed when they are no longer needed, which forces the program to request new frames unnecessarily. Nevertheless, this may not pose a significant issue, given that the benchmarks are designed to limit the overall amount of allocated memory. However, we also save time by omitting deallocations that themselves are not free.

There also might be an effect on the memory allocation algorithms' performance. Since the `std::malloc` version of the benchmarks never deallocates, the allocator can act similarly to a bump pointer. This might give a small advantage to the non-FIP `std::malloc` version of the algorithms. This possible advantage is illustrated in the actual custom bump pointer version of the benchmarks. Even though the FIP bump pointer version still presents performance gains, the difference is noticeably smaller.

Regarding the benchmarking process, a significant portion of the operating system's overhead arises from executing the testing scripts multiple times from scratch. Our benchmarking approach ensures that optimizations within the operating system—such as the allocation of program stacks at different memory addresses for each run do not impact the final results.

6 Related Work

Repeatability and reproducibility are important issues in any research field. Many people have specifically raised these points in relation to systems research [4,10]. Our work contributes to this by reproducing the experimental evaluation of the FIP approach implemented by the Koka language [6,5].

The idea of using a formal system (typically a type system) to track when an in-place update could be safely performed has been used before. Hofmann [3] proposed a linear type system for in-place updates that is very similar to the FIP calculus. The main distinction is that, while the FIP calculus can be integrated with a type system, the calculus itself is type-agnostic.

In general, linear types have often been a consideration for memory-related optimizations. As Wadler [11] pointed out, values belonging to linear types require much simpler memory management and admit destructive modification in a way that is compatible with functional purity.

Linear Haskell [1] is a great example of how linear types can be helpful even if the compiler does not use them to perform optimizations. Haskell libraries often provide unsafe versions of certain primitives (such as `unsafeFreeze` found in various container packages) that place the burden of showing that a copy can be elided or mutation done in-place on the programmer. By using a linear interface, the burden can be placed on the library, where it is much easier to control.

The Koka language deals with the problem of deciding when to use a standard allocating operation or an in-place operation by employing precise reference counting. It might also be possible to perform this decision statically, without the overhead of runtime reference counting. In particular, bounded linear types [2] together with a suitable semiring seem like a promising approach.

7 Conclusion

In-place mutation has long been an advantage of imperative data structures. Although purely functional structures have their own advantages, there are cases where performance is paramount and additional memory allocations required to work with these structures are too steep a price. One of the main culprits is the inability to reuse memory that is no longer needed by the program.

FIP is a promising new approach that tackles this issue in a systematic way. By keeping track of unneeded memory through reuse credits, it is able to execute a wide range of functional programs in-place. However, while Koka, the main implementation of FIP, shows a significant performance speedup, it was not clear how much of it is purely due to the in-place execution.

In this work, we provided a second implementation of the FIP calculus, one specifically made with benchmarking in mind. With it, we reproduced the original experiments and showed that the previously established performance speedup is indeed due to the in-place execution.

While implementing the type system, we noted a potential design limitation of working with reuse tokens: whether a program can be executed in-place can

depend on subtle implementation details. Still, FIP proved to be as powerful as we expected and thus we hope that this work encourages further adoption of this framework.

Acknowledgments. We are grateful to David Bednárek and Jakub Yaghob, authors of the Cecko Skeleton framework that was used as the basis for our implementation. Anonymous reviewers at TFP provided valuable feedback and directions for improving the paper. The work has been supported by the Charles University grant PRIMUS/24/SCI/021.

References

1. Bernardy, J.P., Boespflug, M., Newton, R.R., Peyton Jones, S., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* **2**(POPL) (Dec 2017). <https://doi.org/10.1145/3158093>
2. Ghica, D.R., Smith, A.I.: Bounded linear types in a resource semiring. In: Shao, Z. (ed.) *Programming Languages and Systems*. pp. 331–350. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_18
3. Hofmann, M.: A type system for bounded space and functional in-place update—extended abstract. In: Smolka, G. (ed.) *Programming Languages and Systems*. pp. 165–179. Springer Berlin Heidelberg, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-46425-5_11
4. Krishnamurthi, S., Vitek, J.: The real software crisis: repeatability as a core value. *Commun. ACM* **58**(3), 34–36 (Feb 2015). <https://doi.org/10.1145/2658987>
5. Leijen, D.: The Koka Programming Language (2025), <https://koka-lang.github.io/koka/doc/book.html#why>
6. Lorenzen, A., Leijen, D., Swierstra, W.: FP²: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* **7**(ICFP) (Aug 2023). <https://doi.org/10.1145/3607840>
7. Procházka, J.: Benchmarking a baseline fully-in-place functional language compiler (2025), <https://github.com/JaromirProchazka/FipCompiler>
8. Reinking, A., Xie, N., de Moura, L., Leijen, D.: Perceus: Garbage Free Reference Counting with Reuse. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. p. 96–111. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3453483.3454032>
9. Sansom, P.M., Peyton Jones, S.L.: Generational garbage collection for Haskell. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. p. 106–116. FPCA ’93, Association for Computing Machinery, New York, NY, USA (1993). <https://doi.org/10.1145/165180.165195>
10. Vitek, J., Kalibera, T.: Repeatability, reproducibility, and rigor in systems research. In: *Proceedings of the Ninth ACM International Conference on Embedded Software*. p. 33–38. EMSOFT ’11, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/2038642.2038650>
11. Wadler, P.: Linear types can change the world! In: *Programming concepts and methods*. vol. 3, p. 5. North-Holland, Amsterdam (1990)