# 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 more testable, inherently thread-safe, and easier to reason about. 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 known as fully-in-place functional programming (FIP), which 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 against 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 FIP authors, but it also uncovers limitations of the approach.

**Keywords:** FIP · Compilation · Programming languages · Functional programming

## 1 Introduction

It is no secret that programs written in functional languages allocate a lot of memory. Functional data structures achieve their advantages through immutability. 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 the best for short-lived structures that never leave the first generation.

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 copy 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 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, it is, to our knowledge, 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 if it is possible to obtain the same benefits in a different functional language.

### 1.1 Goals

The primary objective of this work is to design and implement a small, compiled functional language based on the FIP calculus. The implementation should have 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: quicksort, red-black tree insertion, and finger tree insertion. We are interested in the relative speedup between a standard reallocating 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 a potential limitations of the FIP approach.

### 1.2 Contributions

– We implemented StaFip, a lightweight functional language (Section 3). Unlike Koka, StaFip is a statically compiled language without garbage collection. Its feature set was carefully chosen with the goal of providing 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, confirming the validity of the FIP approach (Section 4).
– We observe that the type system can be sensitive to the implementation details (Section 5). In particular, a valid FIP program in one language might not be valid in another language. The degree to which validity depends on implementation details could pose an issue to FIP.

## 2 Background

### 2.1 The FIP Calculus

To introduce the FIP approach, we consider the 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 1.

The FIP approach avoids this inefficiency by reusing the space of the ‘`reversed`‘ list (Figure 1) to initialize the ‘`(_head:accumulator)`‘ node without the need for a new allocation. It is important to keep in mind that `reversed` argument must be owned by (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, let us look at the *reuse tokens*, which are a special notation of FIP calculus developed for Koka and used to trace the available memory. The part ‘`(_head:rest)`‘ is understood as a block of memory with two elements, the head and tail, and is used as a reuse credit of size 2 (written as $\diamond_2$). Upon reuse, it must be checked that the size of the new structure fits the size of the available reuse credit (as seen in Figure 2).

These reuse tokens must represent a contiguous segment of memory. Note that two reuse tokens representing adjacent segments cannot be merged into

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

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

Fig. 1: A Haskell code representing a simple allocating list reversal.

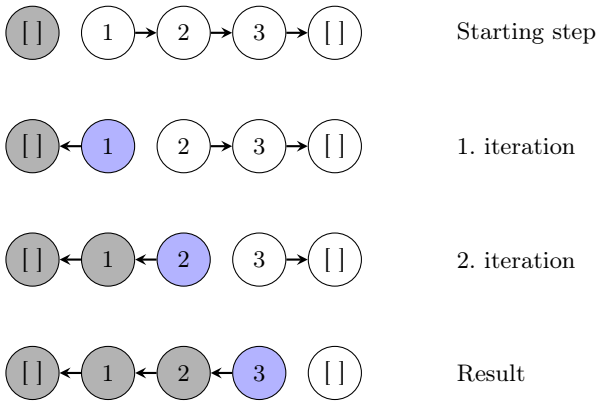



Fig. 2: Step by step flow of the fully in-place list reversal.

a larger reuse token. This is because we cannot ensure that two variables will be allocated right next to each other statically before runtime. The allocation process is managed by the kernel.

### 2.2 The FIP Type System

The original FIP calculus models a richer programming language. Only a subset of the FIP type system is relevant for benchmarking purposes. This includes typing rules for pattern matching on expressions, reuse of available data blocks and calling of first-order functions.

Evaluation Terms:

$$
\begin{array}{lll}
E ::= (V, ..., V) & & \text{(tuple)} \\
\quad \mid \texttt{f}(E; E) & & \text{(call)} \\
\quad \mid \texttt{match}\ E\ \{P \mapsto E\} & & \text{(match)} \\
\quad \mid \texttt{match!}\ E\ \{P \mapsto E\} & & \text{(destructive match)} \\
\\
V ::= x & & \text{(varible)} \\
\quad \mid \texttt{C}\ V\ V\ ..\ V & & \text{(constructor)} \\
\\
P ::= C\ x1\ ...\ xk & & \text{(pattern)}
\end{array}
$$

FIP calculus defines a syntax based on commonly used functional operations: function call, function definition, match and destructive match terms. The match

$$\frac{}{\Delta \mid x \vdash x}\ \textsc{Var} \qquad \frac{\Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma, \diamond_0 \vdash e}\ \textsc{Empty} \qquad \frac{}{\Delta \mid \varnothing \vdash C}\ \textsc{Atom}$$

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

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

terms are used to create branching and unboxing in the program. The destructive match (match!) in particular is a special FIP term which allows the reuse of memory blocks.

For reusing memory, we denote a reuse token of size $k$ as $\diamond_k$. These tokens represent a contiguous part of already allocated memory, all of which is available for reuse. They are created from destroyed data, which the program no longer uses.

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 for any constructor. It would also make it much harder to manage memory, since to deallocate this memory block without destroying some still-used data, the system would have to keep track of all the data instances placed in it. The block could be deallocated only if all the instances placed in it are not used anytime later in the runtime.

Since the FIP approach reuses already allocated memory blocks, converting their data to reuse tokens effectively destroys them, we need to make sure that this data is no longer used by other parts of the program. For that purpose, we use two separate contexts for any expression $e$:

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

$\Delta$ context denotes borrowed environment, with values that are not owned by us, and thus must not be destroyed. $\Gamma$ 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 judgement $\Delta|\Gamma \vdash e$ means that expression $e$ is a well-formed FIP expression in borrowed context $\Delta$ and owned context $\Gamma$. A union of two contexts is denoted by their concatenation: $\Delta, \Delta'$.

*Well-formedness.* A FIP term is well-formed if every allocation reuses memory of a previously destroyed data structure. This is ensured using the well-formedness judgment written as $\Delta \mid \Gamma \vdash e$. Figure 3 shows the base and variable rules. The Var rule states that any expression 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 $\Sigma'$) 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 arguments.

The remaining interestign rules are shown in Figure 4. The Tuple rule states that we can create tuples of variables which are well-formed in the borrowed context and the union of the owned contexts of the variables.

$$\frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash (v_1, \ldots, v_n)}\ \textsc{Tuple} \qquad \frac{\overline{y} \in \Delta, \mathrm{dom}(\Sigma) \qquad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash f(\bar{y}; e)}\ \textsc{Call}$$

$$\frac{y \in \Delta \qquad \Delta, \overline{x}_i \mid \Gamma \vdash e_i \qquad \overline{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma \vdash \texttt{match}\ y\ \{C_i\ \bar{x}_i \mapsto e_i\}}\ \textsc{BMatch}$$

$$\frac{\Delta \mid \Gamma, \overline{x}_i, \diamond_k \vdash e_i \qquad k = |\overline{x}_i| \qquad \overline{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma, x \vdash \texttt{match!}\ x\ \{C_i\ \overline{x}_i \mapsto e_i\}}\ \textsc{DMatch!}$$

$$\frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_k, \diamond_k \vdash C^k\ v_1 \ldots v_k}\ \textsc{Reuse}$$

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 expressions $\bar{x}_i$ are not already in the borrowed or owned contexts. They are new, unique expressions. On the other hand, the DMatch! rule states that an expression is well-formed if the 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 actually introduced to a context. It provides these reuse tokens for the Reuse rule. This rule states that a construction of object $C^k$ using a reuse token of the same size $\diamond_k$ is well-formed if the construction arguments are well-formed.

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

The set of rules defines well-formedness, independently of any particular type system. It is also important to note that realistic programs can not be written using only the FIP syntax. There must be some way of integrating FIP code fragments in a larger non-FIP program. This is because using only the FIP subset, no memory blocks could be allocated, and so no memory blocks could be reused anywhere in the program.

### 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 a language embedded within Koka.

Koka uses reference counting to manage its memory and to ensure the uniqueness of variables 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 for 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 optimizations and static analysis in order to generate code from Koka scripts 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 (like C). It is made to provide a strong baseline for benchmarking the FIP approach as described in the background, without unrelated features from Koka confounding the testing results.

Let us now discuss how the StaFip language represents the type definitions and how the FIP rules are enforced.

### 3.1 Basic StaFip Terms

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

Function definition can either include a body, or simply declare a function without it (this is useful in cyclical dependencies between functions). A function definition consists of an optional `fip` keyword for FIP functions, a return type, a name. a list of explicitly typed parameters. and a its body.

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

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

Fig. 5: Example of main function definition.

### 3.2 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` or a `Cons` with a value and list data fields inside it. The StaFip implementation represents these data types in a similar way to C structs. When using the type as an argument, for instance in '`type list xs`', 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 on Figure 7 with its memory layout in the diagram below the type declaration.

For each of these options a constructor and a reuser functions are implicitly defined. A reuser function resembles a constructor; 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 type in that location, effectively reusing the address. Regarding the FIP theoretical model, this function is used for the consumption of the reuse tokens.

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

This is different for constructor calls with no arguments like the '`Nil`' constructor in the example. The resulting type instances are represented as a `NULL` pointer. Consequently, only one such empty constructor can exist. In our implementation, multiple empty constructors would be indistinguishable from each other at runtime.

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 and pointing to each 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.

### 3.3 Match Expressions

Match expressions is where the magic of the FIP method happens. A match expression is a syntactic structure that, given an input expression, creates branching

```
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 declaration

in the program based on the expression's actual (child/case) type. In StaFip, we distinguish between two types 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 this match in a simple list reversal function (also presented in [6]). In this example, we match an expression `xs`, for which we must specify type `type list` if the `xs` is a `Nil`, this expression is evaluated as the value of variable `acc`. If the expression a list element `Cons(x, xx)`, where we name the element value `x`, and the rest of the list `xx`, this expression is evaluated as value of function call `freverse_acc(xx, Cons(x, acc))`. The type of the matched expression is checked using the type tag allocated on heap in case of the `Cons` type, or by checking it to be NULL in case of the `Nil` type.

But how is the match used for memory block reusing? When a destructive `match!` is applied to an expression whose value is allocated on the heap, this value is tracked by the compiler as reusable. On each return expression, if a type is constructed, instead of calling the type's constructor, a reuser is called, possibly overwriting the matched expression value. 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, the StaFip places this responsibility on the user.

### 3.4 Implementation of Reuse Tokens

When the competition process enters a destructive match or body of a FIP function (function with a key `fip` keyword in its declarators), the compiler switches into a 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 throw a compilation exception or simply allocate instead (weak configuration mostly for debugging). It is important that the FIP mod can be turned off, since in a program with only FIP functions, no memory allocation could 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)

```
type Tree {
    Leaf;
    Node(type Tree left, type Tree right, int key);
}

Node(Leaf, Node(Leaf, Leaf, 2), 1) // instance of Tree
    // 1
    //  \
    //   2
```

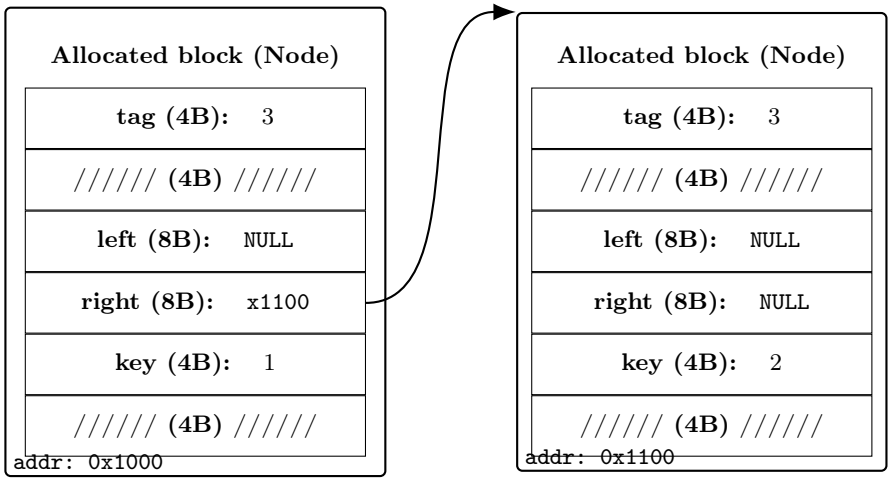



Fig. 7: Binary tree data type example and its heap-allocated instance layout diagram.

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 traced 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 are constructing a type instance 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 used all the reusable blocks, so the context is empty. On line 9, we are at

```
// 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.

another case of the inner match expression. We are outside of the context of the last case, and so the `x1` block is still unreused. 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.

```
1.  match! x1 -> type T1 {          // reusables: emty
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: emty
7.                    c, d          // reusables: emty
8.                )                 // reusables: emty
9.           // | ...                // reusables: x1{3}
10.         }                       // reusables: emty
11. // | ...                        // reusables: emty
12. }                               // reusables: emty
```

Fig. 9: Example of nested match expression

As shown in the example, the reusable blocks tracking is tied to the variable context. Notice how, when the program jumped out of the case context on line 8, where the `x1`'s memory block had already been reused, to the next case's context, the `x1` memory block reappears. This is because we are in 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 context block is entered, for example, on a match or an if-else expression. When a new context block is entered, if 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, the quicksort algorithm, red-black tree insertion, and finger tree insertion algorithms were chosen.

Why these algorithms? Quicksorting 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 much more complex algorithm, more closely aligned with practical applications in functional programming.

These algorithms were also benchmarked in the $FP^2$ paper [6, p. 25]. They were selected to directly compare the efficiency of the garbage-collected Koka with the statistically checked 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. Below each comparison, the relative speedup of the FIP implementation is calculated as (reallocating.*mean* / FIP.*mean*).

As illustrated in Figure 10:

1. **Finger tree insertions** implemented in FIP are around 3x faster than their reallocating implementation in StaFip
2. **Red-Black tree insertions** implemented in FIP are around 2x faster than their reallocating implementation in StaFip
3. **Quick sort** implemented in FIP is around 40% faster than its reallocating implementation in StaFip

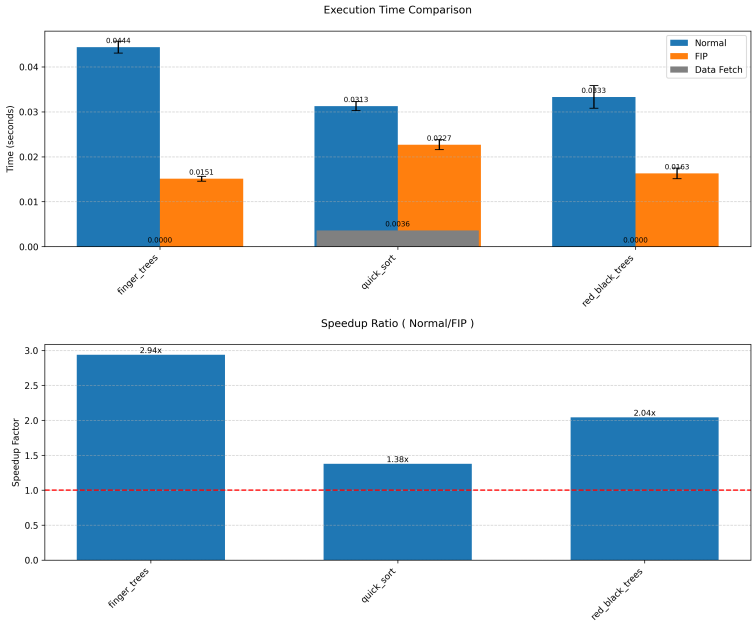
These results, as presented, closely resemble those in the original paper [6, p. 25]. The $FP^2$ paper details their implementation for

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 implementation.




| command | fetch (ms) | mean (ms) | stddev (ms) |
|---|---|---|---|
| finger_trees_normal | 0.0011 | 44.3690 | 1.3114 |
| finger_trees_fip | 0.0017 | 15.0972 | 0.5527 |
| quick_sort_normal | 3.5735 | 31.2762 | 1.0000 |
| quick_sort_fip | 3.5931 | 22.6871 | 1.0911 |
| red_black_trees_normal | 0.0013 | 33.2975 | 2.5120 |
| red_black_trees_fip | 0.0013 | 16.3061 | 1.1851 |

Fig. 10: Results of the benchmarks on Ubuntu 22.04.4 (AMD Ryzen 5 4600HS) in seconds

Consequently, the same algorithm may not be easily translatable into a different language that implements FIP.

This problem is best illustrated on Example 11. In this example, we have a `Tree` and `Buffer` data structures. 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 some heap-allocated block, this algorithm would no longer be fully in-place, since there would be no reuse tokens for it. This is interesting because the FIP calculus does not specify how data types should be implemented. And thus, to functional languages implementing FIP could have algorithms written in them, where the algorithm is not directly translatable to the other language.

```
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. 11: 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.

### 5.2 Behavior to Hardware Caches

Another aspect to consider is how the StaFip programs in general relate to hardware caches. Since FIP programs limit the amount of reallocations and so are inherently nicer to hardware caches. This is because a reallocation of an object often leads to a cache miss when this object is accessed again. This is also one of the reasons why the FIP approach can run much faster than its reallocating counterpart.

### 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 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, we aim to maintain consistency in the implementations, with the only variation being the FIP optimization.

### 5.4 Threats to Validity

Let's now discuss potential aspects of the StaFip testing 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.

Regarding the benchmarking process, a significant portion of the operating systems' overhead arises from executing the testing scripts multiple times from scratch. This approach ensures that optimizations within the operating system—such as allocating program stacks at different memory addresses for each run—do not impact the final results. The benchmarks also assess the display of standard deviations, and the number of runs is adjusted to achieve an appropriately low standard deviation.

## 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 results obtained 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 which 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.

## 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.: $FP^2$: Fully in-Place Functional Programming. Proc. ACM Program. Lang. **7**(ICFP) (Aug 2023). https://doi.org/10.1145/3607840
7. Procházka, J., Petříček, T.: 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)