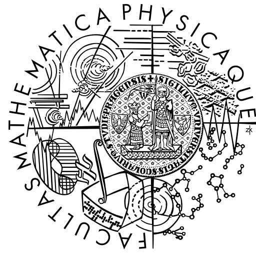



**FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University**

**BACHELOR THESIS**

Jaromír Procházka

# Benchmarking a baseline fully-in-place functional language compiler

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Tomáš Petříček, Ph.D

Study programme: Computer science: Programming and development of software Bc.

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ............. date .............          ....................................

Author's signature

I would like to thank my teachers, Tomáš Petříček, Ph.D., and doc. RNDr. Tomáš Dvořák, CSc., for their support and help with this project, and to RNDr. Bednárek David, Ph.D. and RNDr. Yaghob Jakub, Ph.D., for teaching me the necessary knowledge and providing their useful framework for this work.

Title: Benchmarking a baseline fully-in-place functional language compiler

Author: Jaromír Procházka

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D, Department of Distributed and Dependable Systems

Abstract: Functional languages such as Haskell and F# make code more testable, inherently thread safe and easier to reason about through their functional purity. However, the adoption of immutable data structures introduces 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. Nonetheless, Koka's garbage collection and extensive feature set complicate the task of isolating FIP's specific memory efficiency advantages. The goal of this thesis is to design a minimal functional language called StaFip that supports fully in-place updates utilizing the FIP calculus and omitting garbage collection. This will allow for a comparison of the performance of the FIP approach against that of a conventional implementation for algorithms such as quicksort, red-black tree insertions, and finger tree insertions. The findings of this thesis demonstrate that a language employing the FIP calculus in a garbage collection-free environment can achieve a significant increase in performance and memory efficiency.

Keywords: FIP, Compiler, Optimizing compile, Programming languages, functional programming

Název práce: Návrh a testování FIP funkcionálního jazyka

Autor: Jaromír Procházka

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Funkcionální jazyky jako Haskell a F# činí kód lépe testovatelným, inherentně vícevláknově bezpečným a snáze analyzovatelným díky omezení vedlejších efektů. Nicméně použití neměnných datových struktur přináší náklady z hlediska efektivity, protože mnoho operací vyžaduje kopírování namísto úprav na místě. Z tohoto důvodu jazyk Koka zavedl nový mechanismus zvaný fully in-place calculus (FIP kalkulus), který umožňuje bezpečné úpravy na konstantní paměti a zároveň se vyhýbá zbytečným alokacím. Správa paměti v Koce a jeho bohatá sada funkcionalit však ztěžují odlišení konkrétních přínosů FIP v oblasti úspory paměti. Cílem této práce je navrhnout minimální funkcionální jazyk jménem StaFip, který podporuje fully in-place aktualizace za pomoci FIP kalkulu a který nevyužívá žádnou správu paměti (garbage collection). Jeho výkon testujeme vůči konvenční implementaci algoritmu quicksort, vkládání do černo-červených stromů a prstových stromů. Výsledky ukazují, že jazyk využívající FIP kalkulus v prostředí bez garbage collection může přinést znatelné zvýšení výkonu a úsporu paměti.

Klíčová slova: FIP, Překladače, Optimalizace překladače, Programovací jazyky, funkcionální programování

# Contents


**1 Introduction** **8**
- 1.1 Benefits of Fully in-place approach . . . . . . . . . . . . . . . . . . 8
- 1.2 Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**2 The Fully in-Place Functional Language** **10**
- 2.1 How FIP algorithms work . . . . . . . . . . . . . . . . . . . . . . . 10
- 2.2 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  - 2.2.1 Evaluation of functional languages in general . . . . . . . . 12
  - 2.2.2 Type systems . . . . . . . . . . . . . . . . . . . . . . . . . 12
- 2.3 The FIP Calculus . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  - 2.3.1 Evaluation rules . . . . . . . . . . . . . . . . . . . . . . . . 12
  - 2.3.2 Calculus rules . . . . . . . . . . . . . . . . . . . . . . . . . 13
  - 2.3.3 Store semantics and properties of reduced FIP calculus . . 15
- 2.4 StaFip use of FIP calculus . . . . . . . . . . . . . . . . . . . . . . 16
  - 2.4.1 Uniqueness Typing . . . . . . . . . . . . . . . . . . . . . . 16
  - 2.4.2 Evaluation of StaFip . . . . . . . . . . . . . . . . . . . . . 16
  - 2.4.3 Use of store semantics in StaFip . . . . . . . . . . . . . . . 16

**3 The StaFip Language and Compiler** **18**
- 3.1 Lexical analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
- 3.2 StaFip language . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  - 3.2.1 Declarations . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  - 3.2.2 Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . 21
- 3.3 Syntactic and semantic analysis . . . . . . . . . . . . . . . . . . . 24
- 3.4 Implementation of Semantic Analysis . . . . . . . . . . . . . . . . 25
  - 3.4.1 Declarations . . . . . . . . . . . . . . . . . . . . . . . . . . 26
  - 3.4.2 Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . 30
- 3.5 StaFip library functions . . . . . . . . . . . . . . . . . . . . . . . 34
  - 3.5.1 Object type . . . . . . . . . . . . . . . . . . . . . . . . . . 34
  - 3.5.2 Pair, Tuple3 and Tuple4 types . . . . . . . . . . . . . . . . 36
  - 3.5.3 Boolean type . . . . . . . . . . . . . . . . . . . . . . . . . 36
  - 3.5.4 printf function . . . . . . . . . . . . . . . . . . . . . . . . 37
  - 3.5.5 log function . . . . . . . . . . . . . . . . . . . . . . . . . . 37
  - 3.5.6 time function . . . . . . . . . . . . . . . . . . . . . . . . . 38

**4 The Benchmarks** **39**
- 4.1 Algorithms tested with benchmarks . . . . . . . . . . . . . . . . . 39
  - 4.1.1 Implementation differences between FIP and standard versions 40
- 4.2 Benchmarks generation . . . . . . . . . . . . . . . . . . . . . . . . 45
  - 4.2.1 Project overview . . . . . . . . . . . . . . . . . . . . . . . . 45
  - 4.2.2 Benchmarks overview . . . . . . . . . . . . . . . . . . . . . 45
- 4.3 Result . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
- 4.4 Threats to validity . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**5 Conclusion** **49**
5.1 Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2 The approach summary . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.3 Results overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.4 Future Works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

**Bibliography** **51**

**List of Figures** **52**

**A Attachments** **54**
A.1 Instruction Wrapper Class . . . . . . . . . . . . . . . . . . . . . . . 54
A.2 StaFip grammar . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
A.3 Attachment Archive Structure . . . . . . . . . . . . . . . . . . . . . 62

# 1 Introduction

Functional languages, such as Haskell and F#, provide developers with numerous advantages, including function purity. This purity significantly enhances testability, simplifies the implementation of concurrency, and promotes scalability in software development. However, these benefits come with efficiency drawbacks due to the lack of in-place modifications.

To address this limitation, the Koka language employs a novel mechanism known as the Fully In Place (FIP) calculus [1], which enables safe in-place updates while avoiding unnecessary memory allocations. Nonetheless, Koka's reliance on garbage collection and its extensive feature set may obscure the efficiency gains achieved through the FIP approach.

## 1.1 Benefits of Fully in-place approach

How does Koka maintain its functional purity while performing in-place updates? In functional languages, data types are typically immutable. Consequently, all operations involving these data types necessitate copying rather than utilizing in-place modifications. Koka avoids unnecessary allocations and copying by reusing already allocated variables, which it knows will not be accessed later in the code.

The mechanism can be seen in an example of list reversal (Figure 2.1). The most straightforward method to reverse a list in a functional language involves utilizing an accumulator list as an argument, recursively invoking the reverse function, and appending the head of the reversed list to the accumulator during each call. However, due to the immutability of list nodes, each append operation necessitates the allocation of a new list node that points to the head of the accumulator, necessitating garbage collection of any unused objects created during this process. The Koka compiler optimizes the reverse function by assuming that the provided list is either owned or unique to the reverse function. Each iteration reuses the current head node of the list being reversed and overwrites it with a new head node of the accumulator list. This adjustment transforms constant reallocations into more efficient pointer arithmetic. The entire process is illustrated in the diagram 1.1.

This figure illustrates the memory layout during the execution of the list reversal process, as described above, using the FIP calculus. The white nodes represent the original list, the gray nodes represent the accumulator list and the blue nodes represent the reused (overwritten) node in that step. No new nodes are created here, and at the end of the function, the original list is completely empty and destroyed.

## 1.2 Goals

The goal of this thesis is to design a minimal functional language compiler that supports fully in-place updates utilizing the FIP calculus without the need for garbage collection. Additionally, we evaluate its performance against a conven-

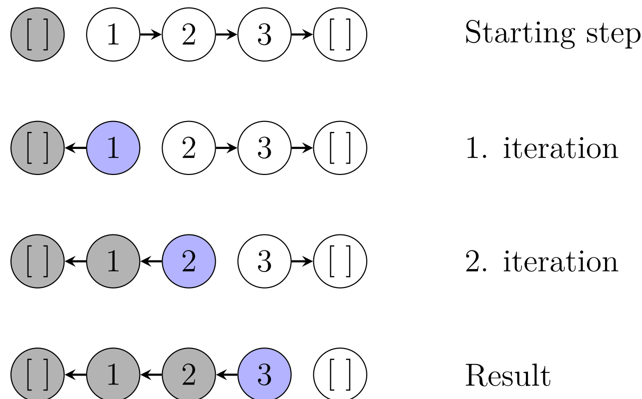



**Figure 1.1** Step by step flow of how the fully in place list reversal looks like.

tional functional approach in a statically checked, compiled environment where each type construction corresponds to a memory allocation. The evaluation focuses on the quicksort algorithm, the red-black tree insertion algorithm—which entails traversal and tree balancing—and the finger tree insertion algorithm. We refer to this language as StaFip.

By implementing both approaches within the same minimalistic language environment, devoid of garbage collection, we eliminate confounding variables that could distort the results. This controlled comparison is particularly pertinent for memory-constrained environments, where the inefficiencies of garbage collection are magnified, making the memory preservation advantages of FIP critical.

This minimal compiler can compile a given code with both FIP optimizations enabled and disabled. In this manner, any performance differences are solely attributable to the optimization itself.

Since garbage collection is least efficient on platforms with limited memory space, and the FIP calculus optimizes memory usage, FIP may offer significant advantages in this context. Moreover, the results of this project demonstrate that the FIP approach yields substantial benefits in terms of performance and memory conservation.

The project's outcomes will evaluate whether FIP-based languages devoid of garbage collection can render functional programming a viable option, particularly in resource-constrained environments.

The purpose of this project can be summarized into a primary goal and two supporting objectives:

1. **A Compiler prototype for a Functional Language** Focusing on front end implementation, with the back end handled by the LLVM library.

2. **Implementation of Common Algorithms** Providing both conventional allocation based and FIP-optimized options.

3. **Benchmark Suite** which is the main goal that will compare these approaches across algorithms and data structures.

# 2 The Fully in-Place Functional Language

The goal of this thesis is to evaluate the performance of the fully in-place (FIP) calculus defined in a 2023 paper by Anton Lorenzen et al. [1]. This work will be referred to as the $FP^2$ paper. The original implementation in Koka uses a dynamic reference counting approach to ensure the uniqueness of reused types. The goal of this work is to test the static evaluation without reference counting. The FIP calculus facilitates the avoidance of unnecessary allocations in our programs. Some key concepts from the $FP^2$ paper relevant to this work will now be explained.

## 2.1 How FIP algorithms work

The prevalent functional languages operate by constantly copying data structures, leading to significant reallocation. In contrast, the FIP calculus avoids these allocations by reusing and overwriting the structures that are already allocated and provided as arguments. To explain the $FP^2$ evaluation further, let us revisit the example presented in the Introduction.

Classical implementation of list reversal in functional languages recursively goes through the list using an accumulator argument and on each item of the list allocates a new node and sets it to the accumulator, as can be seen in Figure 2.1.

The FIP approach solves this by reusing the space of '`reversed`' (Figure 2.1) to initialize the '`(_head:accumulator)`' node without any need for new allocation. But keep in mind that `reversed` argument must be owned by (unique to) the FIP function to keep its functional purity (no side-effects). To understand how the new node can be initialized without allocation, let's look at the *reuse tokens*, which are a special case of Koka used for tracing the available memory. The part '`(_head:rest)`' is understood as a block of memory with two elements, the head and tail, and a reuse credit of size 2 notated as $\diamond_k$. On reuse, it must be checked that the size of the new structure fits the size of the reuse credit available (as seen in Figure 2.2).

These reuse tokens must represent a continuous segment of memory. Two reuse tokens representing segments lying next to each other can not be merged into one larger reuse token.

## 2.2 Evaluation

In this section, the formal model of evaluation of functional languages is described first. Then, it is described how FIP calculus and the StaFip language use this evaluation model. These models are useful for understanding the FIP approach and how the reuse tokens work.

```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


reverse_list :: [a] -> [a]
reverse_list xs = reverse_acc xs []

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

**Figure 2.1** A Haskell code representing a simple not fully-in place list reversal.

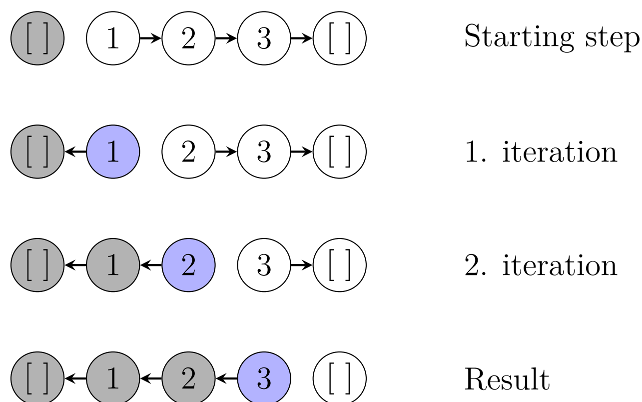



**Figure 2.2** Step by step flow of how the fully in place list reversal looks like.

$$
\begin{aligned}
e &::= \mathtt{f}(e; e) && \text{(call)} \\
&\quad \mid (v, \ldots, v) && \text{(unboxed tuple)} \\
&\quad \mid \mathtt{match}\; e\; \{p \longmapsto e\} && \text{(matching)} \\
&\quad \mid \mathtt{match!}\; e\; \{p \longmapsto e\} && \text{(destructive match)} \\
\text{where} \\
v &::= x, y && \text{(variables)} \\
&\quad \mid \mathtt{C}^k\; v_1 \ldots v_k && \text{(constructor of arity } k\text{)} \\
p &::= \mathtt{C}^k\; x_1 \ldots x_k && \text{(pattern)}
\end{aligned}
$$

**Figure 2.3** Syntax of reduced FIP calculus

### 2.2.1 Evaluation of functional languages in general

Formal models of functional languages are, in general, built on rewriting rules. A functional program is an expression, also called a term. A compiler of a functional program is based on a formal theory of rewriting rules, although they typically do not implement them directly. This theory uses some precisely defined rules in order to repeatedly rewrite the expression of the program until the final resulting expression can not be rewritten anymore. This reduced expression (if it exists) is ideally some literal, like a number. But in principle, any unreduceable expression can be a result of a functional program.

### 2.2.2 Type systems

In theory, the most basic rule set for functional language evaluation is the untyped lambda calculus defined by Alonzo Church in 1930 [2], which was later proven to be Turing complete. However, not all expressions that the lambda calculus can evaluate can be reduced to a final irreducible form, which is called the normal form of an expression. To rule out these expressions in functional languages, a type system is introduced.

A Type system is a set of additional rules that assigns a type to some terms. In the case of lambda calculus, a useful type system gives a type only to terms with a normal form. This typing must also be statically decidable. An example of such a useful theoretical type system is the Hindley-Milner type system [3].

## 2.3 The FIP Calculus

### 2.3.1 Evaluation rules

The syntax rules in Figure 2.3 show a set of (often nested) elements that a FIP functional program is made of. Of course, we are still talking about a theoretical model. These elements are then evaluated using rules in Figure 2.4. In this Figure in the evaluation order section, the hole $\square$ sign represents a space for a term. The $E$ symbol represents a term inside which the evaluation can happen. The *call left*

Evaluation order:

$$
\begin{aligned}
E ::= &\ \square \\
&\mid \texttt{f}(E; e) \ \ \text{(call left)} \\
&\mid \texttt{f}(v; E) \ \ \text{(call right)} \\
&\mid \texttt{match}\ E\ \{p \mapsto e\} \ \ \text{(match context)} \\
&\mid \texttt{match!}\ E\ \{p \mapsto e\} \ \ \text{(destructive match context)}
\end{aligned}
$$

Evaluation steps:

$$
\begin{array}{rlll}
\text{(call)} & \texttt{f}(v_1; v_2) & \to e[y := v_1, x := v_2] & \text{with } \texttt{f}(y; x) = e \in \Sigma \\
\text{(match)} & \texttt{match}\ (\texttt{C}\ v)\ \{p \mapsto e\} & \to e_i[y := v] & \text{with } p_i = \texttt{C}\ y \\
\text{(match!)} & \texttt{match!}\ (\texttt{C}\ v)\ \{p \mapsto e\} & \to e_i[x := v] & \text{with } p_i = \texttt{C}\ x
\end{array}
$$

**Figure 2.4** Evaluation rules of reduced FIP calculus

and *call right* rules say that the first parameter of a function is evaluated first, and then any other parameter can be evaluated only if all the parameters before it are evaluated. A bracket notation is used to substitute a term into the hole $\square$. For instance for $E = \texttt{match!}\ \square\ \{p \mapsto e\}$, expression $E[C\bar{v}]$ is equivalent to $\texttt{match!}\ C\ \bar{v}\ \{p \mapsto e\}$.

### 2.3.2 Calculus rules

The $FP^2$ paper describes evaluation rules and grammar for the whole functional implementation. But for the purposes of benchmarking the FIP implementations, only a subset is actually needed. But for now, let's briefly describe the language for fully in-place updates.

The syntax of FIP calculus closely resembles the classical lambda calculus with the addition of call and destructive match expressions. Call is used for the FIP functions, and the destructive match is a classical match with the reuse of the matched variable.

There are two distinguishable contexts for any expression e. $\Delta$ context denotes borrowed environment whereas $\Gamma$ denotes owned environment. The FIP updates can only be performed on expressions from the $\Gamma$ owned environment. The $\Gamma$ corresponds to the unique typed arguments. The judgement $\Delta | \Gamma \vdash e$ implies that expression $e$ is a well-formed FIP expression in borrowed context $\Delta$ and owned context $\Gamma$. Well-formed FIP expressions satisfy these FIP calculus rules.

The rules are in the form of premises above the line and conclusion below it. For example, the *EMPTY* rule is read like this: "given that $e$ is well formed in borrowed context and owned context (premiss), than it follows that $e$ is well formed in the borrowed context and a new owned context with the reuse token of size 0 and the contents of the owned context (conclusion)". The rules define what expressions are well-formed in the FIP calculus. The Expression is well-formed if there is a chain of application of these rules that results in this expression. The subset of rules that the StaFip language implements can be seen in Figure 2.5.

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

$$\frac{}{\Delta \mid x \vdash x}\ \text{VAR} \qquad \frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash (v_1, \ldots, v_n)}\ \text{TUPLE}$$

$$\frac{\overline{y} \in \Delta, \text{dom}(\Sigma) \qquad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash f(\overline{y}; e)}\ \text{CALL} \qquad \frac{\Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma, \diamond_0 \vdash e}\ \text{EMPTY}$$

$$\frac{}{\Vdash \varnothing}\ \text{DEFBASE} \qquad \frac{}{\Delta \mid \varnothing \vdash C}\ \text{ATOM} \qquad \frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_k, \diamond_k \vdash C^k\ v_1 \ldots v_k}\ \text{REUSE}$$

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

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

$$\frac{\Vdash \Sigma' \qquad \overline{y} \mid \overline{x} \vdash e}{\Vdash \Sigma', f(\overline{y}; \overline{x}) = e}\ \text{DEFFUN}$$

**Figure 2.5** Rules of the reduced FIP calculus.

In practice for the benchmarks, only the *REUSE*, *TUPLE*, *BMATCH* / *DMATCH!*, *CALL*, and some base-case rules are necessary. This is because only in these rules are the reuse tokens used, or are equivalent to rules that use reuse tokens.

The *VAR* rule on Figure 2.5 simply says that any expression in a borrowed environment is well-formed. Then there are the baseline rules like *DEFBASE*, *ATOM*, and *EMPTY* that 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 *TUPLE* rule simply says that we can make tuples of owned expressions. The *DEFFUN* rule shows that a function definition is well-formed if its resulting expression is well-formed in the contexts of the arguments. These defined functions can be used with the *CALL* rule, which says 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.

The rules *BMACH* and *DMACH!* are used to create branching and unboxing in the program. The non-destructive *BMACH* is well formed simply if the matched object $y$ is in borrowed context, each expression $e_i$ matched to 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. The *DMATCH!* rule states, on the other hand, that an expression is well formed if the resulting expression $e_i$ after each 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

$$
\begin{aligned}
S &::= \varnothing \mid S, x \mapsto C^k\, x_1 \ldots x_k \mid S, \diamond_k \\
E &::= \square \mid C^k x_1 \ldots E \ldots v_k \mid (x_1, \ldots, E, \ldots, v_n) \mid \mathtt{f}(\overline{y}; E) \\
&\quad \mid \mathtt{match}\ E\ \{p \mapsto e\} \mid \mathtt{match!}\ E\ \{p \mapsto e\}
\end{aligned}
$$

$$
\begin{array}{llll}
(call_s) & S \mid \mathtt{f}(\bar{y}; \bar{x}) & \rightarrow_s\ S \mid e[\bar{y}/\bar{y}, \bar{x}/\bar{x}] & (\mathtt{f}(y; x) = e \in \Sigma) \\
(reuse_s) & S, \diamond_k \mid \mathtt{C}^k\, x_1 \ldots x_k & \rightarrow_s\ S, x \mapsto \mathtt{C}^k x_1 \ldots x_k \mid x & (k \geqslant 1, \text{ fresh } x) \\
(atom_s) & S \mid \mathtt{C} & \rightarrow_s\ S, x \mapsto \mathtt{C} \mid x & (\text{fresh } x) \\
(bmatch_s) & S, y \mapsto \mathtt{C}^k z \mid \mathtt{match}\ y\ \{p \rightarrow e\} & \rightarrow_s\ S, y \mapsto \mathtt{C}^k z \mid e_i[z/y] & (p_i = \mathtt{C}^k y) \\
(dmatch_s) & S, x \mapsto \mathtt{C}^k z \mid \mathtt{match!}\ x\ \{p \rightarrow e\} & \rightarrow_s\ S, \diamond_k \mid e_i[z/x] & (p_i = \mathtt{C}^k x)
\end{array}
$$

**Figure 2.6** Reduced Store semantics

reuse token is actually provided 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.

### 2.3.3 Store semantics and properties of reduced FIP calculus

Finally, the FP$^2$ paper defines the store semantics that will be used for proving some important properties of FIP programs. It evaluates using a fixed-size store $S$, and its most important property is that no step allocates or deallocates memory. The reduced store semantics can be seen in Figure 2.6.

For the semantics outline in the previous chapter, the $FP^2$[1, sec . 2.3] paper shows that the store semantics is sound for well-formed FIP programs (Theorem 1) and that a FIP program reduces in-place (Theorem 2).

The first theorem shows that the FIP calculus actually can run on the store semantics, given that there are enough reuse tokens. The second shows that for the size of a store $|S|$ defined as:

$$
|S| = \begin{cases} |\emptyset| = 0, \\ |S, \diamond^k| = |S| + k, \\ |S, x \mapsto \mathtt{C}^k x_1 \ldots x_k| = |S| + k. \end{cases}
$$

it hold that $S \mid e \rightarrow_s^* S' \mid e'$ implies $|S| = |S'|$. Or in other words that the stored size does not change throughout the evaluation.

## 2.4 StaFip use of FIP calculus

### 2.4.1 Uniqueness Typing

Since the reused expression is overwritten in FIP functions, it is vital that the structure isn't also used somewhere else in the program. One way to ensure that is by using a Uniqueness-typing system [4] that tracks if the argument is either unique and thus not used elsewhere or shared. The FIP functions are then only allowed to reuse unique types. This can lead to code duplication since we might need to have an implementation for both unique and shared variants of an argument.

The Koka language takes a dynamic approach with reference counting. In this approach, only arguments with reference count equal to one are determined to be unique and thus can be reused. Nevertheless, since the question of whether a structure is owned by the function and safe to be reused is solved statically, and thus has little effect on the benchmarks, solving it will not be a primary goal of this work.

### 2.4.2 Evaluation of StaFip

Although the lambda calculus evaluation is useful in common functional languages, the FIP calculus does not improve on it, and it is not necessary for the testing of the FIPs' performance. StaFip thus implements only a necessary subset of the evaluation rules described in $FP^2$ paper [1].

As can be seen in Figure 2.3, the *application* rule can be ignored because the advantages FIP presents are mostly in reusing constructed structures in memory, not in the implementation of lambda calculus style evaluations with higher order functions. For all function evaluations, we will use the *call* rule. Numerical operations are also a case of *call* rule being applied. The *let* rule can be reduced to *matching rule* since '$\texttt{let}\ \bar{x} = e_1\ \texttt{in}\ e_2$' is equivalent to '$\texttt{match}\ e_1\ \{C^k x_1, ..., x_k \longmapsto e_2\}$' (match with only one pattern) with some constructor $C^k$ defined matching the $\bar{x}$ type.

A compiler implementing FIP ensures that the external behavior of the compiled code is equivalent to the application of a specific sequence of these rules. However, the internal implementation may differ significantly from a straightforward application of these rules. Nonetheless, programmers can utilize these rules to conceptualize the implementation of their programs.

For example, in the application of the described structures, when a function is defined where a certain expression is supposed to generate reusable tokens from a particular expression, we employ a destructive match on it. Although this approach may be lengthy, it provides precise control over which arguments can be reused and which cannot at any branch of the computation.

### 2.4.3 Use of store semantics in StaFip

Since the reduced FIP calculus and store semantics are subsets of the original, with the removed rule for 'let' being equivalent to 'match' with only one pattern, and the removed rule for application having no impact on storage, Theorems 1

and 2 (from section 2.3.3) also apply to the reduced semantics employed by the StaFip compiler.

These properties are significant because they demonstrate that programs, when well-formed in the FIP calculus, do not require any additional data blocks.

# 3 The StaFip Language and Compiler

The StaFip compiler utilizes Flex for lexical analysis [5] and Bison [6] for syntactic analysis. It performs a single traversal, meaning that the input source code is traversed only once before generating the assembly code. The final code is generated using the LLVM library and C++ in the form of LLVM Intermediate Representation (IR), which is subsequently compiled with *Clang* compiler into a final executable file.

The project does not utilize the LLVM library directly; instead, it employs the framework developed by Bednárek and Yaghob, Cecko Skeleton [7]. This framework abstracts the LLVM library and offers additional features, including context management and lookup tables for functions and variables.

## 3.1 Lexical analysis

The language closely resembles the formal definition of evaluation notation in the calculus, as outlined in Chapter 2.4.2 with the addition of specific symbols and explicit typing. The definitions of the lexer tokens are illustrated in Figure 3.1.

In the Flex (/solution/calexer.lex in Figure A.3) file that implements the StaFip lexer, a sequence of regular expressions (REGEX) is provided alongside their corresponding C++ code snippets. This C++ code is designed to generate the defined lexical tokens, throw exceptions for invalid syntax, or update certain global contexts. As the lexer traverses the source code, it attempts to identify the first REGEX expression in the sequence that matches the loaded text of the source code. Upon locating a match, the associated C++ code executes, potentially generating a lexical token. Subsequently, the loaded source code text is cleared, and new source code is loaded character by character, allowing the process to repeat. The output of this lexical parsing comprises a stream of lexical tokens that are subsequently supplied to the syntactic and semantic analysis phase of the StaFip compilation.

Many tokens in the StaFip language draw inspiration from C syntax; it also supports C-style comments such as "`// ...`" and "`/*...*/`" implemented in the lexer. These comments do not generate any tokens.

An important token is the NEWLINE token. It serves to separate function definitions. The token generation is implemented in the Flex file as follows: '`[\n]+ { make_NEWLINE(...)}`' with a rule before it '`[\n][ \t\r]`' which does not create this token. In this way, the `NEWLINE` separator is not generated if some whitespace character that is not a newline precedes the label on the subsequent line. For instance, a function definition must begin at the very start of the line, and its body must be indented with whitespace on the following lines.

Tokens such as `IDF`, `INTLIT`, `TYPEIDF`, and `STRLIT` hold data that can be accessed in syntactic analysis. The MATCH token additionally contains information about its type, whether it is destructive (`match!`) or nondestructive (`match`).

```
EOF         "end of file"

LBRA        "["
RBRA        "]"
LPAR        "("
RPAR        ")"

NEWLINE     "NEW_LINE"
ARROW       "->"
COMMA       ","
AMP         "&"
VERT        "|"
STAR        "*"
ADDOP       "+ or -"
EMPH        "!"
DIVOP       "/ or %"
CMPO        "<, >, <=, or >="
CMPE        "== or !="
DAMP        "&&"
DVERT       "||"
ASGN        "="
SEMIC       ";"
COLON       ":"
LCUR        "{"
RCUR        "}"

TYPEDEF     "type"
ETYPE       "_Bool, char, or int"
SIZEOF      "sizeof"
IN          "in"
LET         "let"
MATCH       "match"
FIP         "fip"
FN          "fn"
IF          "if"
ELSE        "else"

IDF         "identifier"
TYPEIDF     "type identifier"
INTLIT      "integer literal"
STRLIT      "string literal"
```

**Figure 3.1** A lexer tokens list

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

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

**Figure 3.2** Example of list enum type declaration

## 3.2 StaFip language

### 3.2.1 Declarations

#### Functions

As in most functional languages, a program is a set of functions. A function here is a syntactic structure that takes a number of arguments and contains a single expression to which the arguments are substituted. Consequently, a function definition is structured as follows: '`return_type IDF [ argument_list ] = expression`' where the `argument_list` consists of a sequence typically formatted as '`argument_type IDF`' (but the whole definition is '`declaration_specifiers declarator`') separated by commas.

If a function must be declared without a body, for instance in cases where two functions call each other, a C-style notation is utilized: '`return_type IDF [ argument_list ];`'. The body can be declared later in the code using the following notation: '`return_type IDF [ argument_list ] = expression`'.

In the definition of the function, it can also be declared with a fip keyword in its list of specifiers. This notation, similar to that of Koka, indicates that the function is executed fully in place.

When initiating a StaFip program, the compiler searches for a main function in the following format: '`int main [int argc, char** argv] = expression`'. The evaluated value of this function of type int is the exit code of the program.

#### Enum types

StaFip supports custom type declarations. The StaFip types function similarly to Rust's enumerations. A parent type is defined, consisting of a number of child types, each with its corresponding data field. Each child type represents a specific case of the parent type. In StaFip, types are declared with a parent declaration formatted as '`type parent_type_idf { child_type_declaration_list }`' which includes a list of child-type declarations enclosed in curly brackets. A child type is declared as '`child_type_idf( data_fields_list );`'. If the child type has no fields, the parentheses are omitted, resulting in the declaration '`child_type_idf;`'. For reasons outlined in the implementation section, a parent type may have only one child type that lacks data fields. An example of an enumeration-type declaration and instantiation is presented in Figure 3.2.

From the perspective of a C-like implementation, the parent type can be considered an abstract type, with child types deriving from it. The parent type itself cannot be instantiated. Only the child types (or cases) can be constructed

when all their data field values are provided. However, the actual parent type of the child types is statically inferred, and the case for the parent type is determined using match expressions. As illustrated in the `list` type example in Figure 3.2, the second field of the `Cons` child type represents the parent type, which can either be another `Cons` or a `Nil`. A parent type must have at least one associated child type. This guarantees that the field of the parent type always holds an instance of an instantiated child type. To instantiate the type, one must invoke a function with the same name as the type (constructor) as shown: `Cons(1, Nil)`.

There is also an implicitly declared object type: '`type object { Tagged; }`'. This type is unique in that any other type can be cast to it, and it can be cast to any other type. It is important to note that these casts are implicit, and there is no mechanism for explicit casting. This functionality is particularly useful when implementing a generic type, which can take values of different types as fields in various contexts. While this approach may be less safe, it facilitates the easy implementation of a working solution for our testing purposes. However, this generalization to the object type only applies to other declared enum types. Data types such as `int` cannot be cast to the object type.

### 3.2.2 Expressions

#### Literals and Data types

StaFip supports `int`, `string`, `_Bool`, and `char` literals. The `int` is a signed 32-bit integer, `char` is an 8-bit value, and `string` literals are pointers to an array of characters. `_Bool` is a 1-bit value where a literal of 1 represents true, and a literal of 0 represents false. Although C-style pointers are available, their use is not recommended for routine operations.

#### Arithmetics

Arithmetic operations in StaFip are analogous to those in C. For numerical type, StaFip supports addition(+), subtraction(-), multiplication(*), division(/) and modulo(%). Boolean algebra is also supported, encompassing the following operations: greater than (>), less than (<), greater or equil(>=), lesser or equil(<=), equil(==), not equal to (!=) and negation(!).

#### Match expressions

A match expression is a syntactic structure that, given an input expression, creates branching in the program based on the expression's actual (child/-case) type. In the context of FIP, it is also employed to provide reuse tokens to the context, as the size of a matched type in memory is statically known once the type is identified. The syntax of a match expression is `match matched_expression ->expressions_result_type { paterns_list }` with either a non-destructive `BMATCH` (`match`) or a destructive `DMATCH` (`match!`) version. A pattern has the form '`| child_type_idf( new_variables_list )->result_expression`'. The `new_variables_list` is a list of identifiers separated by commas. If a new variable has the same name as another argument above it, the name is overloaded and covered in this context. Each pattern maintains its own context; thus, variable names

```
// type list xs
// type list acc
match! xs -> type list {
     | Nil            -> acc
     | Cons(x, xx)    -> freverse_acc(xx, Cons(x, acc))
}
```

**Figure 3.3** example of match expression on an expression of type list

can be shared between patterns without causing overloading. The field values of the matched expression are copied to the new variables. The `result_expression` can use the new variables from `new_variables_list`, and given that the pattern matches the type of the given matched expression, this expression is the result of the match expression. The figure 3.3 shows an example of match expression use. The '`expressions_result_type`' indicates the type of the evaluated expression of the match expression and must be explicitly defined. Each `result_expression` must conform to this explicitly specified type.

The matched expression can be any expression. However, if no pattern corresponds to the resulting type of the matched expression, the behavior is considered undefined.

Match expressions are present in Koka and other programming languages. The limitation of the StaFip's match is that the matched expression must be of an Enum type. Consequently, numbers, strings, and similar types cannot be pattern-matched. Additionally, the pattern can only use identifiers as variables, excluding the use of other patterns. So pattern like '`| Sublist(a, Sublist(b, xs1))->...`' is not possible, and what must be done is nesting the match expressions like this '`| Sublist(a, ss)-> match ss ->type list { |Sublist(b, xs1)->... }`' However, for our testing purposes, this limitation is acceptable because it only slightly complicates the examples, and they can always be expressed in StaFip.

### If-Else expressions

Another option for branching in the program is the if-else expression with this syntax '`if (cond_expression)then_expression ; else else_expression ;`'. The expression in the `cond_expression` must be convertible to a boolean.

Enum types are also Boolean convertible. If-expression given an expression of the enum type as condition enters the `then_expression` if the expression is an instance of the child type with some data inside it, like for example `Cons (5, Nil)`, the result of the if-else expression is the `then_expression`. Otherwise, if the expression is an instance of a type with no data inside it, like `Nil`, the `else_expression` is the result.

The if-else expressions can also be nested as shown in Figure 3.4. This nesting is interpreted similarly to that in the C language. When evaluating the nested if-else expressions, the StaFip compiler traverses the structure until it reaches the deepest if expression that itself has no other nested if/if-else expressions. The next closest else expression is interpreted as belonging to this if expression. This if-else expression is interpreted as the inner expression of the if-else expression above it in the nesting, and this process is repeated. For a more detailed understanding of how the nested if-else expression is interpreted, study the end of figure A.2.

```
// part of the quick sort algorithm implementation
if (_elem(p) <= _elem(x))
    if (_elem(p) <= _elem(y))
        Pair(Nil_elem, Cons_elem(x, Cons_elem(y, Nil_elem)));
    else
        Pair(Cons_elem(y, Nil_elem), Cons_elem(x, Nil_elem));
else
    if (_elem(p) <= _elem(y))
        Pair(Cons_elem(x, Nil_elem), Cons_elem(y, Nil_elem));
    else
        Pair(Cons_elem(x, Cons_elem(y, Nil_elem)), Nil_elem);
```

**Figure 3.4** Example of nested If-Else expression. It results in some instances of Pair type.

The limitation of the If-else expression in StaFip is that its result type must be an Enum type. So, for example, this code is invalid in StaFip '`if (cond)1; else 2;`' since 1 and 2 are of data type, not enum type.

## 3.3 Syntactic and semantic analysis

Using tokens from the lexical analysis, the compiler defines a StaFip grammar. This grammar is visualized in Figure A.2. It is a Context-free Grammar from which the Bison environment builds an automaton called *canonical collection*. For more information on Bison parser generator, reference this manual [6]. This automaton is at the core of the parser that then generates the final Intermediate Representation (IR) of the code using the LLVM C++ library from a stream of lexical tokens from the lexer.

The default class of Context-Free Grammar employed by Bison, as well as by the StaFip compiler, is the LALR(1) grammar. The "LA" part denotes a subtle division of the grammar class, and its consequence is, among other how the next grammar rule is chosen in cases when there are multiple possible options. The "L" signifies the direction of input reading, which, in this context, is from left to right (R refers to reading from right to left). The next R part denotes the production of the Right derivation by the grammar (L for left derivation). Finally, the number 1 represents the look-ahead distance. This means that while deciding how to reduce, the compiler sees only one next terminal (token) ahead. The derivation tree is constructed in a bottom-up manner using the production rules. The input program is deemed valid if, at the end of the token stream, a derivation tree is formed with the `translation_unit` non-terminal at its top.

These properties allow for an effective way to traverse the token stream only once while generating the IR instructions. The LALR(1) grammar is effective as it balances the number of tables needed for parsing with the parsing speed. Using the grammar, the compiler generates a *canonical collection*. To understand it, let's first define some terms. An *Item* in LR(0) grammar $G = (T, N, S, P)$ is a production rule with a special dot symbol ($\bullet$) on the right side ($\mathrm{A} \longrightarrow \alpha \bullet \beta : \alpha, \beta \in (N \cup T)^*; A \in N$). This symbol indicates the current position in the parsing process. A *Closure* of the set of items $I_i$ is a set of items containing $I_i$ such as $\forall \mathrm{A} \longrightarrow \alpha \bullet \mathrm{B}\beta$ in the closure of $I_i$ where $B \in N$ and $\forall \mathrm{B} \longrightarrow \gamma$, item $\forall \mathrm{B} \longrightarrow \bullet\gamma$ also lies in the closure of $I_i$.

The *canonical collection* of LR(0) grammar has these closures as states. It starts with the closure of the items of the rules with the start symbol on the left side and the dot ($\bullet$) symbol on the leftmost side of the right-hand side of the rule ('$A \rightarrow \bullet\omega : A \in N, \omega \in (N \cup T)^*$). Edges with tokens go between states. For each token that is in some item in the current state directly followed by the dot, an edge with that symbol goes to the closure of the item of the same rule, but with the dot moved one place to the right. These transitions effectively describe all possible next symbols in any state of the parsing. If a token is encountered such that there is no transition for it in this state machine, it results in a syntax error.

However *canonical collection* of LR(0) grammar is limited, and conflicts may arise between rules while parsing. For that reason, there is the look-ahead of 1 in the *canonical collection* of LR(1) which prevents these conflicts. More thorough explanations and examples are in Douglas Thain's book [8, p. 51].

Due to this implementation, it is essential for every grammar, at any point during compilation, to unambiguously imply the possible next tokens that could appear in syntactically correct code, as well as those tokens that would result in an immediate syntax error. Additionally, the rules should be applicable with only a lookahead of one token. The StaFip grammar satisfies these constraints. If this requirement is not satisfied in the grammar, Bison can be configured to generate an output with some examples of the conflicts and sequences of tokens that create these conflicts (Bison manual [6]).

## 3.4 Implementation of Semantic Analysis

For each rule in the grammar (figure A.2), there exists a C++ code segment responsible for generating the final Intermediate Representation (IR) instructions utilizing the LLVM library. This code is placed in the `/solution/caparser.y` Bison file in the FipCompiler project in appendix A.3. When a rule is selected during the derivation (syntax tree generation), the corresponding code block is invoked.

In Bison, a non-terminal can hold one instance of data. When entering code block of a rule, the data of the right-hand side of the rule can be accessed and used for further IR generation. Also, new data can be stored on the left-hand side of the non-terminal rule. The right-hand side data of the rule can be accessed using the following notation: '`$k`' where k is the index of the token on the right side, starting from 1 (0 is an index of the left side non-terminal). New data can be stored to the left side using this notation: '`$$ = //...data`'. If no code block is present for a rule, the default action is '`$$ = $1;`' (where the data from the first token is forwarded to the non-terminal on the left-hand side).

```
postfix_expression:
    IDF LPAR argument_expression_list RPAR      {
        $$ = casem::handle_postfix_expression_fcall(ctx, $1, $3);
    }
```

Note that data can only be sent upward to the left side non-terminal. However, in some cases, we might need to pass some data from one left-hand side token to the right-hand side.

For instance, we might need to pass type information to a variable of a function declared with that type. This can be solved elegantly using a functional approach. Instead of sending a variable down the syntax tree to another right-hand side token, we can pass a lambda function with all the needed actions from the right-hand side token upward to the problematic rule, and there call the lambda with the present type information. In StaFip implementation, this approach is used in the function and argument declarations.

### 3.4.1 Declarations

#### Enum types

Enum types declaration in StaFip can be represented 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 number. On each new instantiable type declaration, a tag is chosen for it as the least not yet taken positive integer. Also for this type, a constructor '`_const_<type_name>`' and a reuser '`_reuse_<type_name>`' are defined. A reuser function resembles a constructor; however, instead of allocating memory on the heap, it accepts a pointer to a heap location given as its first argument and initializes the type in that location, effectively reusing the address. An example of these can be seen in figure 3.5, which shows an implementation of the list enum type example in figure 3.2. One of these implicitly defined functions is invoked during instantiation as follows: `Cons(1, Nil)`.

Both of these functions take the explicitly declared field values of the type and use a pointer to the heap block of the enum type size to set the tag value and fields. Finally, they both return this pointer. The difference between these functions is how they obtain the heap pointer. While the constructor allocates memory for the required type-sized block, the reuser accepts an additional first argument: a pointer to already allocated memory, where it initializes the type instance.

This is different for child types that with no explicit fields like the '`Nil`' enum type in the example. These types are represented as a `NULL` pointer. Consequently, only one child of such a type can exist. In our implementation, multiple empty enum child types would be indistinguishable at runtime.

An intriguing challenge encountered while developing this testing compiler was the representation of empty enum types. Initially, these types were represented as memory-allocated blocks containing only the tag field. However, this posed a problem later when testing the FIP algorithms since at least the algorithms shown in the $FP^2$ paper [1] did not account for reusing these empty types. There simply were not enough reuse tokens for them. To make our implementation truly fully in place, these types could not be implemented as memory allocated blocks. This shows how FIP can be implementation sensitive, and it might not be easy to migrate algorithm implementations from one FIP language to another.

Another interesting consequence of FIP is the uniform field size. Since each field is one element in a reuse token. Therefore, we must for some types reserve more space than they actually need. Let's illustrate this on these two types: '`Nums(int x, int y);`', and '`Ptrs(type Nums n1, type Nums n2);`'. Since `int` has a size of 4 bytes and type `Nums` has two of them, it takes up 8 bytes. The `Ptrs` type holds two 8-byte pointers and thus takes up 16 bytes. But the `Nums` type should actually be reusable for the `Ptrs` type. But, implemented naively, `Nums` does not hold enough space for it. For this reason, we need to pad the `Nums` type so that any type with 2 fields can reuse it. In the StaFip implementation, for any type allocation, we allocate `number_of_field * ptr_size` bytes, plus the tag field, as no field can exceed the pointer size in the StaFip implementation.

Finally, there are also implicitly declared library types. These types include

the parent type of all enum types called `object` and its child type `Tagged`. Then there are special types like `Pair`, `Tuple3`, `Tuple4`, which can hold several instances of any type (defined with `object` type parameters). These types are special because their construction/reusing does not allocate on the heap and thus they do not need any reuse tokens. These types are used for returning multiple instances from a single function.

There is also a `Boolean` implicitly declared type which simply holds an int instance representing a boolean. These standard library types are declared in the compiler using the `casem` library function `declare_support_functions()`.

```
%Cons = type { i32, i32, ptr }
%list = type { i32 }

define ptr @_const_Nil() {
start:                                                  ; preds = %
   prolog
  ret ptr null
}

define ptr @_const_Cons(i32 %0, ptr %1) {
prolog:
  %val = ; first arg
  %next = ; second arg
  %_result = ; result variable
  br label %start

start:                                                  ; preds = %
   prolog
  %2 = call ptr @ckrt_malloc(i32 24)
  store ptr %2, ptr %_result, align 8
  %"$_result" = load ptr, ptr %_result, align 8
  %_result._tag = getelementptr inbounds %Cons, ptr %"$_result",
     i32 0, i32 0
  store i32 6, ptr %_result._tag, align 4   ; 6 is the tag for
     Cons type
  %"$_result1" = load ptr, ptr %_result, align 8
  %_result._tag2 = getelementptr inbounds %Cons, ptr %"$_result1"
     , i32 0, i32 0
  %"$_result3" = load ptr, ptr %_result, align 8
  %_result.val = getelementptr inbounds %Cons, ptr %"$_result3",
     i32 0, i32 1
  %"$val" = load i32, ptr %val, align 4
  store i32 %"$val", ptr %_result.val, align 4
  %"$_result4" = load ptr, ptr %_result, align 8
  %_result.next = getelementptr inbounds %Cons, ptr %"$_result4",
      i32 0, i32 2
  %"$next" = load ptr, ptr %next, align 8
  store ptr %"$next", ptr %_result.next, align 8
  %"$_result5" = load ptr, ptr %_result, align 8
  ret ptr %"$_result5"
}
```

```
define ptr @_reuse_Cons(ptr %0, i32 %1, ptr %2) {
prolog:
  %"@ru" = ;first arg
  %val = ; second arg (first field val)
  %next = ; third arg (second field)
  %_result = ; result variable
  br label %start

start:                                            ; preds = %
   prolog
  %"$@ru" = load ptr, ptr %"@ru", align 8
  store ptr %"$@ru", ptr %_result, align 8
  %"$_result" = load ptr, ptr %_result, align 8
  %_result._tag = getelementptr inbounds %Cons, ptr %"$_result",
     i32 0, i32 0
  store i32 6, ptr %_result._tag, align 4
  %"$_result1" = load ptr, ptr %_result, align 8
  %_result._tag2 = getelementptr inbounds %Cons, ptr %"$_result1"
     , i32 0, i32 0
  %"$_result3" = load ptr, ptr %_result, align 8
  %_result.val = getelementptr inbounds %Cons, ptr %"$_result3",
     i32 0, i32 1
  %"$val" = load i32, ptr %val, align 4
  store i32 %"$val", ptr %_result.val, align 4
  %"$_result4" = load ptr, ptr %_result, align 8
  %_result.next = getelementptr inbounds %Cons, ptr %"$_result4",
      i32 0, i32 2
  %"$next" = load ptr, ptr %next, align 8
  store ptr %"$next", ptr %_result.next, align 8
  %"$_result5" = load ptr, ptr %_result, align 8
  ret ptr %"$_result5"
}
```

**Figure 3.5** Example of abreviated LLVM IR code generated for list enum type shown in figure 3.2. Note how in `@_const_Cons` at star, malloc() is called while in `@_reuse_Cons` the pointer is given as a function argument.

### 3.4.2 Expressions

#### Literals, Data types and Arithmetics

For literals, arguments, and other data type instances, StaFip compiler uses the `InstructionWrapper` class shown in Figure A.1. The whole class is immutable, and each operation produces a new instance (like `operator!` function, which negates boolean type convertible values). Most operations simply create a new instance of `InstructionWrapper` without changing the old one. Writing a compiler with a language without garbage collection is quite cumbersome, and in our context, the better performance possible with C++ does not matter since the speed of the generated code is the important requirement, not the compilation speed. For these reasons, StaFip implementation mainly stores instances of these structures on the stack and uses copy constructors when necessary. If needed, C++ smart pointers are used to handle memory management.

`InstructionWrapper` class also provides the arithmetical operators that may generate load instructions of the values of the data based on their type, and generates the arithmetical operation instruction using the LLVM library. These functions, in general, also implicitly try to cast given values to their expected types for that operation. If this cast fails, a proper error message is generated and logged, and an invalid `InstructionWrapper` is past instead.

This class also implements operations for enum types, like field parsing, generating an operation for checking if a given enum type has a certain field, and also loads, stores, and debugging prints instructions generation.

#### Match expressions

Match expressions create branching in the code. Figure 3.6 shows the C representation of the match expression evaluation implementation. Of course, the StaFip compiler does not generate C but the LLVM IR code. However, the LLVM IR is a bit hard to understand even for such a small example, and so we provide a C equivalent code. The actual IR representation can be found in the StaFip project repository.

(To obtain the file, first run the benchmarks as described in the README file. The IR representation can be found in the StaFip compiler projects folder in the 'compiled_programs_data' folder in a sub-folder of the name corresponding to the sought script's name in a text file as shown in appendix A.3.)

First, the result variable is declared, which ultimately holds the value of the match expression upon completion of the evaluation. In the StaFip compiler, it is represented by the `InstructionWrapper` class and is propagated up the syntax tree as a result of the `match_expression` non-terminal defined in the grammar presented in figure A.2.

As mentioned in the language section 3.2.2, any pattern that matches an empty type without explicit fields (such as the Nil type) must be positioned as the first pattern within the match patterns. As illustrated in the figure 3.6, this requirement arises because, prior to dereferencing the matched expression and checking its tag, it is essential to confirm that the matched expression is not `NULL`. For a match with an empty type as the first pattern, we generate a branch conditioned on the matched expression being `NULL`. A positive outcome

of this condition will direct control to the block that handles the empty-type pattern expression. The evaluated expression is then stored in the designated result location.

In the negative outcome of the branch, the program jumps to a block where the tag checks start. There a chain of branchings for each Enum type with a pattern is generated. If the tag of the matched expression matches the Enum type tag, the corresponding evaluated expression is stored in the result value, and the checks continue. This chain is equivalent to a sequence of if-statements, each checking the tag and, on success, storing its corresponding evaluated expression to the result location.

Finally, the result location is interpreted as the evaluated match expression and used further in the code.

If the match is destructive, as in the example, the FIP mode is started in the compiler, and the match provides its matched expression as a reuse token. The reuse token from the matched expression goes to the `casem::FipState` singleton which puts it to its internal stack. This class is described in detail in the section 3.4.2. The subsequent expressions in patterns are evaluated with reusers, which may use the matched expression instead of constructors, as seen in the example C code Figure 3.6. This FIP mechanism is described in detail later.

In case of nested match expressions (or even if-else expressions), a new context block is entered so that multiple result locations can coexist and be referenced separately.

While developing the StaFip compiler match expression, the match had to be modified to accommodate the types represented with `NULL` pointers, a problem mentioned in the Enum types subchapter 3.4.1.

Another interesting problem that arose with the match expression implementation was the unintended entering of multiple patterns. In a non-FIP context, any matched expression will result in entering only one of the cases (patterns), given that each pattern matches a unique type. And so, although a bit less performant, the tag checks can be implemented with a sequence of `if` blocks instead of a sequence of `if`/`else if` blocks. This is because the matched expression must have some concrete tag that will not change, and given that each pattern checks for a different type, at most one will check for the tag present in the matched expression.

But when introducing FIP to a functional language, this assumption fails because the matched expression can be reused for instance, of a different type. If any subsequent pattern then checks for this new type, it will be unexpectedly entered, and the result location will be overwritten. This is a problem, and it was actually encountered in some benchmarked algorithms. The good news is that this problem can be easily solved in the benchmarked algorithms by rearranging the patterns to a different order, such that the subsequent patterns do not check for the type the matched expression could have been reused for. Although this is unpleasant for writing in StaFip, it is enough for our testing purposes. A real-world implementation of StaFip should, instead of a sequence of `if` blocks (the IR equivalent of this higher-level description) generate a sequence of `else if` blocks (`if` - `else if` - `else`) for each case. This way, it is clear that for each match, at most one block implementing the case is entered.

```c
struct list *xs = /*...*/;
struct list *acc = /*...*/;
struct list *_result;

if (!xs) {                      // type Nil is represented as NULL
    _result = acc;
}
else {
    if (xs->tag == 6) {         // 6 is tag value for type Cons
        int x = xs->val;
        struct list *xx = xs->next;

        _result = freverse_acc(xx, _reuse_Cons(xs, x, acc));
        // destructive match reuses xs for Cons construction
    }
}
// _result is forwarded holding the evaluated match expression value ->
```

**Figure 3.6** C representation of the destructive match expression from figure 3.3 evaluation code generated by the StaFip compiler.

### If-Else expressions

The if-else expression is implemented in a similar way to the match expression. At the start of its evaluation, a result variable is declared and then passed forward as the value of the if-else expression. Unlike in a match expression, however, the result type isn't known at the start of the expression, and so we declare the result location with the object type. This is the reason why the result type of these expressions must be an enum type, since other data types can not be cast to it.

After the result type declaration, a conditional jump is made based on the value of the boolean convertible condition expression. Both the `if.then` and `if.else` blocks generated store to the result location and then jump to `if.end` block where the code continues.

### FIP mechanism implementation

The FIP context is handled through the `casem::FipState` singleton (Figure 3.8). Functions '`FipState::enter_fip_mode()`' and '`FipState::exit_fip_mode()`' handle statically at compilation time if the program is in the FIP mode or not. They are implemented with a counter, which is incremented on entry and decremented on exit function. If it is zero, we are not in the FIP mode; otherwise, we are. This information can be retrieved using '`FipState::is_in_fip_mode()`' function. The enter function is called at the start of any expression that uses FIP, like a destructive match expression or a body entry of an FIP function, and the exit function is called when its evaluation is done.

In the compilation process, this class is first used for setting the compiler into a FIP state. This is done in the '`function_definition_head` rule in the Grammar Figure A.2 where the function sets the FIP state if it has '`fip`' keyword in

```
match! x1 -> type T1 {                    // reusables: x1{3}
    | Ta(a, b, c) -> match! x2 -> T1 {// reusables: x1{3}, x2{2}
        | Tb(d, e)  -> Taa( Tbb(a, b), c, d ) // reusables: emty
        // ...                            // reusables: x1{3}, x2{2}
    }                                     // reusables: x1{3}
    // ...
}
```

**Figure 3.7** Example of nested match expression

its head's prefix. It is then again turned off when exiting the function in the 'function_definition rule. In the 'match_head', where the FIP state is entered if the match is destructive (`match!`). And again turned off (decrementing the counter) when exiting the match expression in the 'match_expression' rule.

Let us first illustrate the idea of reuse tokens tracking on an example of nested destructive match expression in Figure 3.7.

At the deepest expression `Taa( Tbb(a, b), c, d )`, the construction of type `Taa` should reuse the expression `x1`, and the construction of type `Tbb` should reuse the expression `x2` because of their sizes. Therefore, the expressions `x1` and `x2` should be available for reuse at this point. But the expression `x2` certainly should not be reusable from this destructive match in other patterns of the outermost match expression. For this purpose, we track the available reuse tokens through different contexts, and each pattern creates its own.

The `casem::FipState` statically keeps track of the reuse expressions in the program at compile time. Inside this class is a context stack of type '`std::vector< std::set<ReuseInstruction, ReuseInstructionComparator>>`'. The `std::vector<>` implements the stacks of contexts. The context is represented as a set `std::set< ReuseInstruction, ReuseInstructionComparator>` with wrapped `InstructionWrappers` representing the reusable heap blocks ordered by the number of their reuse tokens.

As mentioned, 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 `std::set<ReuseInstruction, ReuseInstructionComparator>` context instance 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.

Functions on the `FipState` singleton '`FipState::emplace_reusable(Args &...args )`' and '`FipState::insert_reusable(ReuseInstruction &inst)`' are used for adding new reusable tokens to the current context at the top of the stack.

Using this mechanism, we statically collect and keep track of reuse tokens throughout the program. when evaluating an expression with type initializations in FIP mode, the StaFip compiler uses the '`FipState::reuse(std::size_t required_size)`' to retrieve the reuse token (reusable expression) of a certain size from the current context stored at the top of the `FipState` stack. If there is a token of given size available, the function removes it from the context and instead of the constructor generates a reuser function with this expression in the first argument, as can be seen in Figure 3.6 in the `freverse_acc` function call argument.

Let's apply this to the `generate_ttype_construction` function directly. This behavior is implemented through the `generate_ttype_construction` function in Figure 3.9. This function is called in the '`postfix_expression`' rule (in the '`casem`

```
class FipState
{
    FipState();
public:
    static FipState *GetFipState();

    void enter_fip_mode();
    void exit_fip_mode();
    bool is_in_fip_mode();

    template <class... Args>
    void emplace_reusable(Args &...args);
    void insert_reusable(ReuseInstruction &inst);
    ReuseInstruction reuse(std::size_t required_size);
    void enter_fip_context(bool exiting_function = false);
    void exit_fip_context();
};
```

**Figure 3.8** A header of the FipState class handling the reuse tokens.

`::handle_postfix_expression_fcall()`'). If the postfix expression is a construction call, the `generate_ttype_construction` is used and there as shown in Figure 3.9, it is first decided if the compiler is in FIP state, and if the constructed type is represented by a heap block. If so, the FIPState singleton is used for retrieving a suitable reuse token as described above, and a reuser function call is returned. Otherwise, if the compiler is not in the FIP state, a simple constructor function call is returned. Notice that runtime does in no way decide the reusing, everything is decided at compile time.

If a reusable token of suitable size is not available, two things can happen. The StaFip compiler provides a static variable '`ENFORCE_FIP`' which is by default true. If it is true, then the lack of available tokens results in a compilation error. If it is false, then a constructor with allocation is generated instead without an error. This is a weaker implementation of the FIP, but it can be more practical. The Koka implementation also allows this construction on no-reuse tokens of the required size.

## 3.5 StaFip library functions

StaFip also implements a set of predefined types and functions, often encapsulating some side effect, like, for example, printing to standard output. These functions are then used in the implementation of algorithms for debugging and monitoring the runtime of the StaFip using scripts.

### 3.5.1 Object type

The StaFip standard library includes a definition of an Object type with a child type `Tagged`. This type is provided for purposes of polymorphism. Every

```
InstructionWrapper generate_ttype_construction(
    cecko::context *ctx,
    cecko::CIName &label,
    casem::InstructionArray params)
{
    cecko::CIName constructor_name;
    auto type_rtoken_count = find_ttype_size(label);
    if (fip_state.is_in_fip_mode() && is_capeable_of_reusing(label))
    {
        auto &&to_be_reused = FipState::GetFipState()
            .reuse(type_rtoken_count);
        if (!to_be_reused.valid)
        {
#if ENFORCE_FIP
            ctx->message(cecko::errors::SYNTAX, ctx->line(), fip_fail_msg);
#else
            constructor_name = get_constructor_label(label);
#endif
        }
        else // to_be_reused is valid
        {
            params.insert(params.begin(), to_be_reused.reuseable);
            constructor_name = get_reuser_label(label);
        }
    }
    else
    {
        constructor_name = get_constructor_label(label);
    }

    return init_instruction_function_call(
        ctx,
        init_instruction_from_name(ctx, constructor_name),
        params
    );
}
```

**Figure 3.9** StaFip internal function implementing the type construction behavior.

```
type object {
    Tagged;
}
```

**Figure 3.10** The definition of the object type.

```
fip type Tuple3 buncons [type bseq bs]
    = match! bs -> type Tuple3 {
        | BSeq(s, b) ->
            match uncons(s, b) -> type Tuple3 {
                | Tuple4(x, u3, s_, b_) ->
                    Tuple3(x, u3, BSeq(s_, b_))
            }
    }
```

**Figure 3.11** Example use of `Tuple3` library type. Notes that although the `buncons` function is FIP and there is only one reuse token of size 2 (from `bs` parameter), the `Tuple3` type is constructed and returned since it does not need any reuse tokens.

other defined enum type is implicitly substitutable into a parameter of `type object` (or `type Tagged`). There is an implicit cast from any type to `type object` and from it to any type. The implicit definition of this enum type is visualized in Figure 3.10.

### 3.5.2 Pair, Tuple3 and Tuple4 types

```
Pair(type object first, type object second);
Tuple3(type object first, type object second, type object third);
Tuple4(type object first, type object second, type object third,
   type object forth);
```

These types are only meant for returning multiple instances from a function to a caller without a need for reuse tokens in case of the FIP functions. Regarding the implementation, at the start of a StaFip script, a *field* in memory is allocated for each of these types. When constructing these types later in a script, they simply reuse their corresponding *field*. This can, in general, result in overwriting of still-used data. But since these types are in the testing scripts, only used in cases of returning from a function (and unboxing of the returned instance by the caller), and since each script is only single-threaded, these *fields* are overwritten only when no other function is using them. An example of `Tupple3` type is shown in Figure 3.11.

This implementation is not ideal for the production-ready implementation of StaFip, since in a multi-threaded environment, this implementation might create race conditions. A better implementation would entail storing instances of these types on the stack and copying them to the caller.

### 3.5.3 Boolean type

```
Boolean(int value);
```

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

type list second [int x, type list xx] = xx

int travers_list [type list xs] = match xs -> int {
    | Nil                -> printf("Nil\n")
    | Cons(x, xx)        ->
        travers_list( second( log(x, "-> "), xx ))
}

type list log_list [type list xs] = second( travers_list(xs), xs)
```

**Figure 3.12** Example of a library log function being used to implement a simple log_list function which given a list prints all of its elements in order.

The Boolean type is an implicitly defined enum type used for returning a Boolean type instance from a function to the caller. As the Tuple types, it uses a statically allocated heap *field* which is reused at each construction of `type Boolean` type. This type is defined because the `match` expression must return an enum type instance, and in some instances, we want a function to return a `_Bool`. For this purpose, we make the function return through a `match` expression an instance of `type Boolean`.

### 3.5.4 printf function

```
int printf [char * format, ...]
```

This function is implemented basically the same way as in the C language. It is a variadic function that takes a string with a number (or none) of substrings in format '`%c`' where `c` is some char or sequence of chars identifying type, and a number of arguments. The string representations of the arguments are in order attempted to be substituted by the pattern where the '`%c`' type must match the type of the argument. Also, the number of arguments must match the number of the '`%c`' patterns in the string. A more detailed explanation can be found in the C++ reference documentation [9].

### 3.5.5 log function

```
T log [T e, char *msg = ""]
```

This functions as another function for printing runtime values to standard output. However, unlike the `printf`, which returns the object, the object `e` it was given to it as an argument.

The `T` type can either match `type Object` or any other primitive type like an int or a string. If the given type is a string, it also prints its address in memory. After that, the `msg` string argument is printed. In Figure 3.12 is an example of a list printing implementation using the `log` method.

```
static clock_t clock_start = 0;

void ckrt_measure_cpu_time(void)
{
    if (clock_start == 0)
    {
        clock_start = clock();
    }
    else
    {
        clock_t end = clock();
        double elapsed_sec = (double)(end - clock_start)
            / CLOCKS_PER_SEC;
        printf("Now already malloced in total %uB\n",
            (unsigned int)already_allocated);
        printf("Elapsed CPU time: %.6f\n", elapsed_sec);
        clock_start = 0;
    }
}
```

**Figure 3.13** C++ function implementing the StaFip time function.

### 3.5.6 time function

`T time [T e]`

This function is used for benchmarking and measuring the CPU time of a given part of code. The parameter here is irrelevant; it is simply forwarded to the caller. The resulting printed text contains the measured time in seconds and the amount of memory bytes allocated so far in the program's runtime. On its first call, it starts a clock and on its second call, prints the clock value to the stdout and resets the clock so that this can be repeated again. The object 'e' given to it is simply forwarded to the caller. This clock function is implemented using C code in Figure 3.13.

The function can be used to measure the time a function '`int A [int a]`' takes to finish like this: '`time(A(time(5)))`'. The first call of time on the last int parameter 5 of `A` constructor starts the clock, and right after that the fucntion '`A`' is entered. Then, when fucntion `A` finishes, and its result is given to the second `time` call, the clock is stopped by the `time` function and the measured time is printed to standard output.

# 4 The Benchmarks

## 4.1 Algorithms tested with benchmarks

For benchmarking purposes, the quicksort algorithm, red-black tree insertion, and finger tree insertion algorithms were chosen.

The quicksort algorithm is one of the earliest sorting algorithms, with an average time complexity of $O(n \log(n))$. Quicksort operates by selecting a 'pivot', dividing the input into three lists of elements, those smaller than the pivot, those equal to the pivot, and elements greater than the pivot. It recursively sorts these shorter lists and then concatenates them into a single sorted list [10, p. 259]. Furthermore, it splits and traverses a given list, which is an area where fully in-place programming excels. Sorting algorithms serve as classic benchmarks 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. A red-black tree is a balanced binary tree that marks its nodes as either red or black. It is used to quickly find an inserted element with some ordering. During the insertion process, the new element is initially placed into the tree as a leaf. Subsequently, balancing rules and edge rotations are applied to ensure that the time complexity of the find operation remains $O(\log(n))$ [10, p. 206]. The time savings afforded by the FIP occur primarily during the balancing and traversal phases, rather than during the insertion itself. This test case was chosen due to the requirement to allocate a new element and the necessity of traversing a complex data structure. Moreover, it employs a frequently utilized data structure.

Finally, the finger tree insertion represents a much more complex algorithm, more closely aligned with practical applications in functional programming. A finger tree is a functional data structure representing a sequence of elements similar to a list. However, unlike a functional linked list, it readily exposes the last element in its structure, in addition to the first. Consequently, the *tail* operation exhibits constant time complexity, in contrast to the time complexity of linked lists $O(n)$. Furthermore, it offers improvements to the $find$ operation [11, p. 200]. Like red-black trees, allocation is required for each new element added. However, the FIP optimization becomes relevant subsequent to this allocation.

These algorithms were also benchmarked in the $\text{FP}^2$ paper. 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.

### 4.1.1 Implementation differences between FIP and standard versions

#### red-black tree

In the FIP implementation of the red-black tree, we first traverse the current structure to identify the appropriate position in the binary tree for the new value. Then the tree is balanced using edge rotation with the method *balance*.

The traversal is conducted using a zipper, a prevalent pattern in fully in-place programming. For each type of traversed item, a corresponding type of the same size is created to maintain the traversal state. As the structure is traversed, each item along the path is reused by an instance of the zipper of the same size. When the algorithm backtracks—either at the end of the traversal or when dealing with tree structures—any time a leaf is encountered, the zipper items along the path are "zipped up" by reusing the zipper items based on the type present in the tree prior to the start of the traversal. Upon completion of the traversal, the tree remains unchanged from its initial state at the beginning of the program. The implementation of the traversal is illustrated in Figure 4.1.

The balancing is then implemented in a standard way as described in the Cormen et al. [12, p. 316]. The StaFip implementation is shown in Figure 4.2.

The non-FIP implementation does not use the zipper pattern. In the process of tree traversal and balancing, simple recursion is used with calls to constructors as shown in Figure 4.3.

#### quick sort

In the functional Merge Sort algorithm, the input list is divided into sublists, each containing a single element. These sublists are inherently sorted due to their singular length. Subsequently, these sublists are iteratively transformed into longer, sorted sublists until a fully sorted list is achieved. To facilitate this process, a zipper data structure is employed. This structure is illustrated in Figure 4.4.

This structure is then used in the quick sort algorithm.

#### finger tree

Using finger trees fully in place is, under the standard definition, not feasible due to the varying sizes of the sub-tree instances' types. This issue is addressed by padding the types with integers, ensuring that all types have a uniform size of three, as illustrated in Figure 4.5. Additionally, there are instances where the construction of a finger tree requires extra memory cells. This is resolved by pairing the finger tree sequence with a buffer of size three, which serves solely as temporary storage for values when necessary. For further details, reference the $FP^2$ paper [1, p. 21].

It is noteworthy that when working with FIP algorithms, unnecessary padding is frequently added to enhance program performance. However, these tricks can render the code more difficult to comprehend and often result in increased length.

In the absence of a non-FIP implementation, padding is not utilized, resulting in fewer cases in the algorithm matche expressions.

```
// RB tree
type tree {
     Node(int _color, type tree lchild, type Integer key, _Bool
         value, type tree rchild);
     Leaf;
}

// ZIPPER for rb tree
type accum {
     Done;
     NodeL(int _color, type accum lchild, type Integer key, _Bool
         value, type tree rchild); // has same size as tree.Node
     NodeR(int _color, type tree lchild, type Integer key, _Bool
         value, type accum rchild); // has same size as tree.Node

     // The NodeL and NodeR denote if in the traversal at that
         point the algorithm went Left of Right
}

// Insertion helper function
fip type tree ins [type tree t, type Integer key, _Bool v, type
    accum z]
     = match! t -> type tree {
          | Leaf                   ->
              balance(z, Node(1, Leaf, key, v, Leaf))
              // found leaf, traversal done, start balancing
          | Node(c, l, kx, vx, r) ->
              if (_int(key) < _int(kx))
                   ins(l, key, v, NodeL(c, z, kx, vx, r));
                   // reusing Node for the zipper
              else
                   if (_int(key) > _int(kx))
                        ins(r, key, v, NodeR(c, l, kx, vx, z));
                        // reusing Node for the zipper
                   else
                        balance(z, Node(c, l, key, v, r));
     }
```

**Figure 4.1** Fip implementation of the red-black tree zipper traversal at the start of the insert algorithm.

```
fip type tree balance [type accum z, type tree t] = match! z ->
   type tree {
     | Done                              -> rebuild(z, t)
     | NodeR(c1, l1, k1, v1, z1) ->
        if (c1 == 1) match! z1 -> type tree {
             | Done ->
                Node(0, l1, k1, v1, t)
             | NodeR(c2,l2,k2,v2,z2) ->
                if (is_red(l2))
                    balance(z2, Node(1, set_black(l2), k2, v2,
                       Node(0, l1, k1, v1, t) ));
                else
                    rebuild(z2, Node(0, Node(1,l2,k2,v2,l1), k1,
                       v1, t));
             | NodeL(c2,z2,k2,v2,r2) ->
                if (is_red(r2))
                    balance(z2, Node(1, Node(0,l1,k1,v1,t), k2,
                       v2, set_black(r2) ));
                else
                    match! t -> type tree {
                        | Node(c, l, k, v, r) ->
                            rebuild(z2, Node(0, Node(1,l1,k1,v1,l
                               ), k, v, Node(1,r,k2,v2,r2)))
                    };
        };
        else
             rebuild(z, t);
     | NodeL(c1, z1, k1, v1, r1) ->
        if (c1 == 1)  match! z1 -> type tree  {
             | Done ->
                Node(0, t, k1, v1, r1)
             | NodeL(c2,z2,k2,v2,r2) ->
                if (is_red(r2))
                    balance(z2, Node(1, Node(0, t, k1, v1, r1),
                       k2, v2, set_black(r2) ));
                else
                    rebuild(z2, Node(0, t, k1, v1, Node(1,r1,k2,
                       v2,r2)));
             | NodeR(c2,l2,k2,v2,z2) ->
                if(is_red(l2))
                    balance(z2, Node(1, set_black(l2), k2, v2,
                       Node(0,t,k1,v1,r1) ));
                else
                    match! t -> type tree {
                        | Node(c3, l, k, v, r) ->
                            rebuild(z2, Node(0, Node(1,l2,k2,v2,l
                               ), k, v, Node(1,r,k1,v1,r1)))
                                                   };
        };
        else
             rebuild(z, t);
}
```

**Figure 4.2** FIP implementation of the red-black tree balancing at the end of the insert algorithm.

```
type tree ins [type tree t, type number k, _Bool v]
    = match t -> type tree {
        | Leaf -> Node(1, Leaf, k, v, Leaf);
        | Node(c, l, kx, vx, r)   ->
            if (c == 1)
                if (_int(k) < _int(kx))
                    Node(1, ins(l, k, v), kx, vx, r);
                else
                    if (_int(k) > _int(kx))
                        Node(1, l, kx, vx, ins(r, k, v));
                    else
                        Node(1, l, k, v, r);
            else
                if (_int(k) < _int(kx))
                    if (is_red(l))
                        balance_left(ins(l,k,v), kx, vx, r);
                    else
                        Node(0, ins(l, k, v), kx, vx, r);
                else
                    if (_int(k) > _int(kx))
                        if (is_red(r))
                            balance_right(l, kx, vx, ins(r,k,v));
                        else
                            Node(0, l, kx, vx, ins(r, k, v));
                    else Node(0, l, k, v, r);
    }
```

**Figure 4.3** Non-FIP implementation of the red-black tree insertion operation helper function. Uses simple recursion and calls to constructors, resulting in allocation.

```
type sublist {
    SCons(type elem a, type sublist cs);
    STuple(type elem a, type elem b);
    // sublist with at least 2 elements
}
type partition {
    Sublist(type sublist c, type partition bdl);
    Singleton(type elem c, type partition bdl);
    End;
}

//Cons(4, Cons(3,Cons(2,Cons(1,Nil)))) ->
//Singleton(4, Singleton(3, Singleton(2, Singleton(1, End)))) ->
//Sublist(STuple(3,4),Sublist(STuple(1,2),End)) ->
//Sublist(STuple(1,SCons(2,STuple(3,4))),End) ->
//Cons(1,Cons(2,Cons(3,Cons(4,Nil))))
```

**Figure 4.4** Definition of the zipper for sorting functional algorithms and an example of use in merge sort.

```
// NON-FIP finger tree
type afew_object {
     One(type object a);
     Two(type object a, type object b);
     Three(type object a, type object b, type object c);
}
type seq_object {
     Empty;
     Unit(type object a);
     More(type afew_object l, type seq_object s, type afew_object
        r);
}

// FIP finger tree
type afew_object {
     One(type object a, int _b, int _c);
     Two(type object a, type object b, int _c);
     Three(type object a, type object b, type object c);
}
type tuple_object {
     PairObject(type object a, type object b, int c);
     TripleObject(type object a, type object b, type object c);
}
type seq_object {
     Empty;
     Unit(type object a, int _b, int _c);
     More0(type object l, type seq_object s, type afew_object r);
     More(type tuple_object l, type seq_object s, type afew_object
         r);
}

     // FIP buffered finger tree
     type buffer {
          BNil;
          BCons(type buffer next, int _b, int _c);
     }
     type bseq {
          BSeq(type seq_object s, type buffer q);
     }
```

**Figure 4.5** Definition of the normal finger trees at the top and the FIP finger tree and buffered finger tree at the bottom.

## 4.2 Benchmarks generation

### 4.2.1 Project overview

The StaFip implementation is integrated within a fork of the "Cecko Skeleton" framework developed by Bednárek and Yaghob for educational purposes [7]. As previously mentioned, it employs Bison[6] and Flex[5] for semantic analysis and parsing, respectively.

The project encompasses a `solution` folder that contains the StaFip implementation, which serves as the core of this thesis. The project's structure is illustrated in Attachments A.3. The contents of this folder represent the contributions of this thesis. Additionally, an infrastructure exists for generating benchmarks, tests, and documentation. For benchmarks, the `bench` folder contains the benchmarked custom scripts, while the `test` folder includes the compiler tests along with a set of Bash and Python scripts located in the `benchmarks_utils` folder.

### 4.2.2 Benchmarks overview

For running the benchmarks with the algorithms described above, the StaFip project, as detailed in the attachment A.3, contains a '`generate_fip_tests_data.sh`' file at its root. This script fully automates the execution of benchmarks and the generation of figures. The specifications and usage of this tool are outlined in the project's README and accompanying documentation. This script executes all tests in the `bench` directory located at the root of the project, which are designated as benchmark algorithms. The aforementioned algorithms are automatically compiled and executed repeatedly on moderately sized data sets. This includes sorting a large list of numbers and constructing red-black trees and finger trees. The results are stored in the '`compiled_programs_data`' folder at the root and include the graph figure and the table presented in the 'Results' section below in Figure 4.6.

The time measurement is retrieved directly by the script. In StaFip compiler, there is a library function `T time [T e]`. The expression 'e' provided to this function is forwarded to the caller and may be of object type or any primitive type. This clock function is implemented in C, as shown in Figure 3.13.

This function is invoked immediately before and after the execution of the tested algorithms on the designated data. This implementation ensures that the measured time accurately reflects only the duration of the algorithm's execution. The time taken by the operating system to load the script and perform related tasks is excluded from this measurement.

Some algorithms, such as quicksort, necessitate time to prepare their testing data. For instance, quicksort must initially create the extensive list that it will sort. Consequently, the benchmarking program records two measured times: one for data preparation (designated as `fetch_time`) and the other for the actual time taken by the algorithm under test.

## 4.3 Result

The benchmark results are presented in Figure 4.6. They are visualized in both a table and a graph. The table enumerates the results for each algorithm, including both FIP and non-FIP (or "normal" versions. The results are as follows:

1. **fetch_time:** As described in the previous section, the time it took to prepare the testing data for the algorithm in seconds
2. **mean:** the expected value approximation of the algorithm's runtime in seconds
3. **stddev:** the standard deviation approximation of the algorithm's runtime in seconds

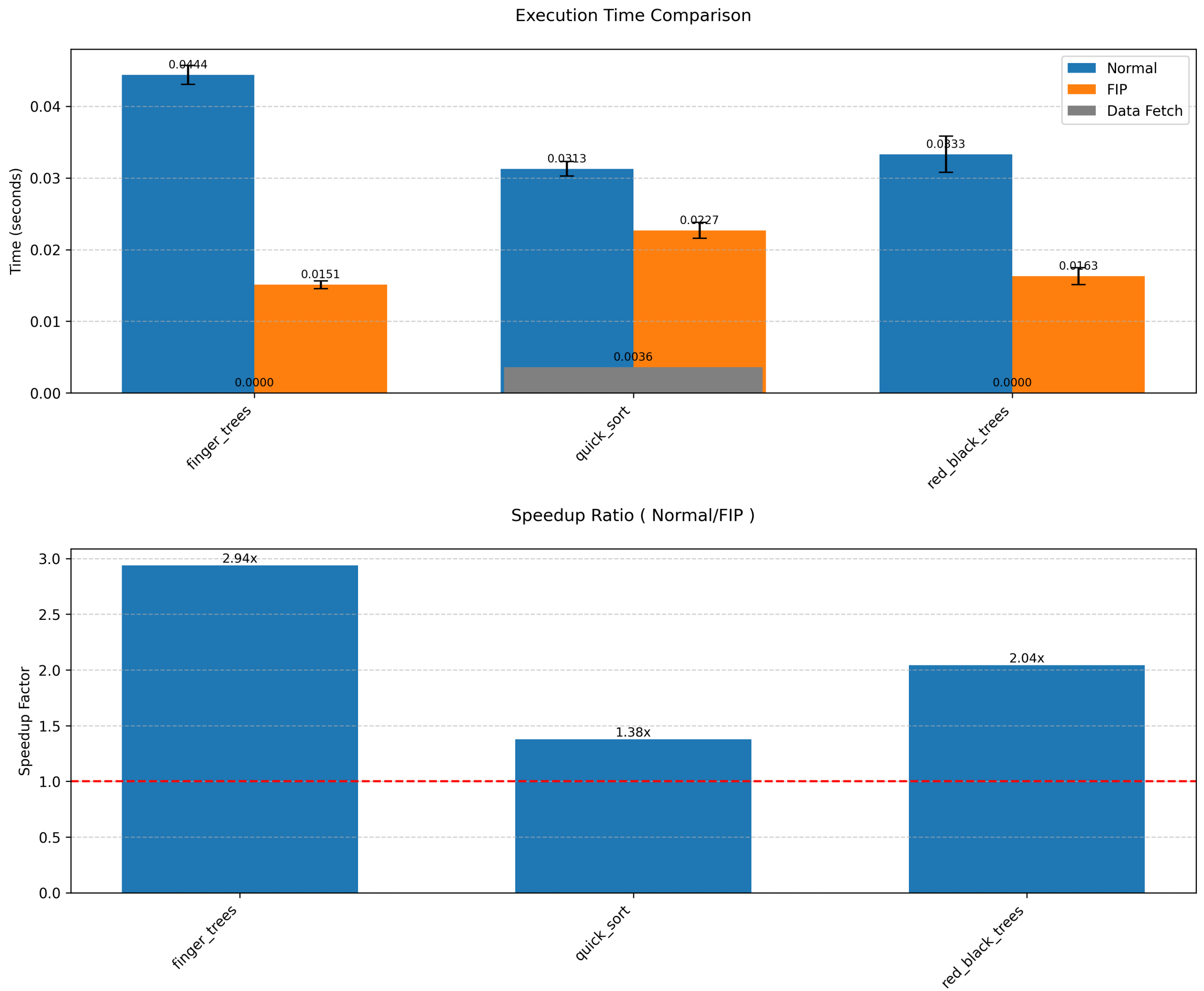
The graph visualizes these results through two rows of column graphs. The upper row for each algorithm compares the means of the FIP (orange) and non-FIP (blue) versions. Above these comparisons, the fetch time is depicted in gray. Below each comparison, the relative speedup of the FIP implementation is calculated as (non-FIP.*mean* / FIP.*mean*).

Regarding the results, as illustrated in Figure 4.6:

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

These results, as presented, closely resemble those in the original paper [1]. Moreover, the absence of runtime garbage collection and other advanced functionalities contributes to their improved performance. 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




| command | fetch_time | mean | stddev |
|---|---|---|---|
| finger_trees_normal | 0.0000011 | 0.0443690 | 0.0013114 |
| finger_trees_fip | 0.0000017 | 0.0150972 | 0.0005527 |
| quick_sort_normal | 0.0035735 | 0.0312762 | 0.0010000 |
| quick_sort_fip | 0.0035931 | 0.0226871 | 0.0010911 |
| red_black_trees_normal | 0.0000013 | 0.0332975 | 0.0025120 |
| red_black_trees_fip | 0.0000013 | 0.0163061 | 0.0011851 |

**Figure 4.6** Results of the benchmarks on Ubuntu 22.04.4 (AMD Ryzen 5 4600HS) in seconds

## 4.4 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 primary concern with the testing process is the absence of a deallocation mechanism. For the FIP implementation within the compiler, this issue is minimal. This is because the FIP algorithm neither allocates memory nor requires deallocation. However, for the non-FIP algorithm, this can result in excessive page faults in the operating system. These page faults may occur more frequently, as the allocated heap blocks are not freed when no longer needed, forcing 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 memory allocated. The `time` library function used in the benchmarks tracks not only the CPU time spent but also the total amount of memory allocated. Furthermore, the testing data for the benchmarks are structured to ensure that the overall memory usage remains within reasonable limits.

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 non-FIP 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.

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.

# 5 Conclusion

## 5.1 Goals

As stated in the introduction, the objective of this thesis is to develop a prototype compiler for a functional language that implements the FIP front-end optimization. The result is the StaFip Compiler, which is included in the attachments A.3. This documented experimental compiler includes a suite of automated verification tests and benchmarks.

This leads us to the second objective of this thesis: implementing standard testing algorithms in the StaFip language. Within the StaFip compiler project, the `bench` folder contains the implementations of the quicksort algorithm, the insertion algorithm for red-black trees (which encompasses traversal and balancing steps), and the finger trees algorithm. Each of these algorithms is non-trivial; for each, both a non-FIP version and an FIP version have been developed.

Finally, the objective is to create a set of benchmarks for the algorithms compiled by the StaFip compiler. The results of these benchmarks are presented and discussed in the Results section 4.3. The algorithm scripts are executed multiple times to measure the mean and standard deviation of their CPU run times. Even considering the challenges outlined in the threads-to-validity section (4.4), the results indicate that FIP optimization offers a significant performance advantage.

## 5.2 The approach summary

The StaFip Compiler implements the functional StaFip language described in the section StaFip language 3.2. This compiler is a small experimental prototype developed using the LLVM library in conjunction with the bison and flex tools. The Cecko skeleton [7] served as the primary framework for the project, with the implementation detailed in the StaFip implementation section 3.4. To facilitate the FIP functionality, there is a `casem::FipState` singleton that tracks heap blocks eligible for reuse by other allocations throughout the compilation process. Additionally, it provides an interface to search for a suitable heap block for reuse.

The standard algorithms are based on the implementations presented in the original paper, but are written in StaFip. This approach facilitates a more direct comparison of this work with the results of the $FP^2$ paper. The FIP rules entail additional assumptions, such as the requirement that the pattern-matched expression must have a type of the same size as the corresponding result type. Due to these assumptions, FIP can significantly increase the complexity of the code, necessitating techniques such as data type padding or the use of zippers.

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. Consequently, the same algorithm may not be easily translatable into a different language that implements FIP. For instance, if, in one implementation, the empty type `Nil` is represented by a heap-allocated block, while in another implementation, this type is equivalent to `NULL`, migrating the algorithm from the second language

to the first can pose challenges. This is due to the requirement in the first language for reuse tokens associated with the `Nil` type, which are not necessary in the second language.

Finally, the benchmarks are described in the Benchmarks Generation section 4.2 and utilize a StaFip library function `time`, which prints the measured CPU time of specific portions of the currently running script to standard output. The benchmarking script executes the target script multiple times, capturing and analyzing the printed CPU time. Subsequently, the script generates a table and graphs displaying the mean and standard deviation of the times as the primary output. This process is fully automated and can be replicated by installing the StaFip project included in the attachments and re-running the benchmarking script outlined in the `README` file.

## 5.3 Results overview

The benchmark results are comparable to those presented in the original $FP^2$ paper [1] and indicate that the FIP optimization positively influences performance. Moreover, these findings demonstrate that the FIP approach outlined in the original $FP^2$ paper is replicable, which is crucial given the current replication crisis in computer science.

However, certain aspects of StaFip may lead to inflated results. The absence of a memory-free operation implementation can cause excessive page faults. This is, however, mitigated by limiting the total amount of allocated memory and by many repetitions of the experiment in a benchmark.

Based on the results from both works, it is implied that numerous functional languages could benefit from the implementation of FIP, thereby reaping the advantages of the functional approach, such as a reduction in side effects, enhanced parallelization, improved testability, and increased code reusability, all while maintaining the superior performance of in-place operations. However, to fully leverage FIP optimization, various techniques and novel patterns must be utilized. This can lead to longer and less readable code.

## 5.4 Future Works

Regarding potential improvements and future work to further test and implement this approach, the development of a language with more advanced features and the implementation of uniqueness semantics may be beneficial. While functional languages that do not employ garbage collection are relatively uncommon, they could find applications in memory-constrained environments, such as embedded systems.

# Bibliography


1. Anton Lorenzen, Daan Leijen; Swierstra, Wouter. $FP^2$: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages*. 2023, vol. 7, no. 198, pp. 275–304.
2. Church, Alonzo. An unsolvable problem of elementary number theory. *American Journal of Mathematics*. 1936.
3. Hindley, J. Roger. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*. 1969, vol. 146, pp. 197–217.
4. Haller P.; Odersky, M. *Capabilities for uniqueness and borrowing*. 2010. Available also from: `https://lampwww.epfl.ch/~phaller/doc/capabilities-uniqueness2.pdf`.
5. The Flex Project, The Regents of the University of California. *Lexical Analysis With Flex, for Flex*. 2012. Available also from: `https://westes.github.io/flex/manual/`.
6. Free Software Foundation, Inc. *Bison*. 2021. Available also from: `https://www.gnu.org/software/bison/manual/html_node/index.html`.
7. Bednárek D., Yaghob. *Cecko Skeleton*. Charles University, Prague, 2024. Available also from: `https://gitlab.mff.cuni.cz/teaching/nswi098/cecko/skeleton`.
8. Thain, Prof. Douglas. *Introduction to Compilers and Language Design*. 2020. Second Edition. isbn 979-8-655-18026-0.
9. cplusplus. *cplusplus*. Available also from: `https://cplusplus.com/reference/cstdio/printf/`.
10. Martin Mareš, Tomáš Valla. *Průvodce labyrintem algoritmů*. CZ.NIC, z. s. p. o, 2022. Second Edition. isbn 978-80-88168-66-9.
11. Ralf Hinze, Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*. 2006, vol. 16, no. 198, pp. 197–217.
12. Cormen, Thomas. *Introduction to Algorithms*. Massachusetts Institute of Technology, 2009. Third edition. isbn 978-0-262-03384-8.

# List of Figures


1.1 Step by step flow of how the fully in place list reversal looks like. . 9

2.1 A Haskell code representing a simple not fully-in place list reversal. 11
2.2 Step by step flow of how the fully in place list reversal looks like. . 11
2.3 Syntax of reduced FIP calculus . . . . . . . . . . . . . . . . . . . 12
2.4 Evaluation rules of reduced FIP calculus . . . . . . . . . . . . . . 13
2.5 Rules of the reduced FIP calculus. . . . . . . . . . . . . . . . . . . 14
2.6 Reduced Store semantics . . . . . . . . . . . . . . . . . . . . . . . 15

3.1 A lexer tokens list . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.2 Example of list enum type declaration . . . . . . . . . . . . . . . 20
3.3 example of match expression on an expression of type list . . . . . 22
3.4 Example of nested If-Else expression. It results in some instances of Pair type. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.5 Example of abreviated LLVM IR code generated for list enum type shown in figure 3.2. Note how in `@_const_Cons` at star, malloc() is called while in `@_reuse_Cons` the pointer is given as a function argument. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.6 C representation of the destructive match expression from figure 3.3 evaluation code generated by the StaFip compiler. . . . . . . . 32
3.7 Example of nested match expression . . . . . . . . . . . . . . . . 33
3.8 A header of the FipState class handling the reuse tokens. . . . . . 34
3.9 StaFip internal function implementing the type construction behavior. 35
3.10 The definition of the object type. . . . . . . . . . . . . . . . . . . 36
3.11 Example use of `Tuple3` library type. Notes that although the `buncons` function is FIP and there is only one reuse token of size 2 (from `bs` parameter), the `Tuple3` type is constructed and returned since it does not need any reuse tokens. . . . . . . . . . . . . . . . . . . . . 36
3.12 Example of a library log function being used to implement a simple log_list function which given a list prints all of its elements in order. 37
3.13 C++ function implementing the StaFip time function. . . . . . . 38

4.1 Fip implementation of the red-black tree zipper traversal at the start of the insert algorithm. . . . . . . . . . . . . . . . . . . . . . . 41
4.2 FIP implementation of the red-black tree balancing at the end of the insert algorithm. . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.3 Non-FIP implementation of the red-black tree insertion operation helper function. Uses simple recursion and calls to constructors, resulting in allocation. . . . . . . . . . . . . . . . . . . . . . . . . 43
4.4 Definition of the zipper for sorting functional algorithms and an example of use in merge sort. . . . . . . . . . . . . . . . . . . . . 43
4.5 Definition of the normal finger trees at the top and the FIP finger tree and buffered finger tree at the bottom. . . . . . . . . . . . . . 44
4.6 Results of the benchmarks on Ubuntu 22.04.4 (AMD Ryzen 5 4600HS) in seconds . . . . . . . . . . . . . . . . . . . . . . . . . 47

A.1 An instruction wrapper class used to represent general run-time expressions and operations on them . . . . . . . . . . . . . . . . . . 55
A.2 The StaFip Grammer where translation_unit is the root nonterminal (%start) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

# A Attachments

## A.1 Instruction Wrapper Class

```
class InstructionWrapper
{
InstructionWrapper();
InstructionWrapper(cecko::context *vctx, VarMode vmode,
    cecko::CKIRValueObs instruction, cecko::CKTypeSafeObs vtype, bool
    vconst, const cecko::CIName &vname = "t");

void generate_debug_print(const cecko::CIName &label);
void generate_print(const cecko::CIName &label);
void generate_print(InstructionWrapper &label);
bool is_valid();
InstructionWrapper operator*() const;
InstructionWrapper operator&() const;
InstructionWrapper store(const InstructionWrapper &other);
cecko::CKIRValueObs get_ir() const;
cecko::CKIRValueObs read_ir(cecko::CKTypeSafeObs vtype) const;
cecko::CKIRValueObs read_ir() const;
InstructionWrapper operator+(const InstructionWrapper &other) const;
InstructionWrapper operator-(const InstructionWrapper &other) const;
InstructionWrapper operator-() const;
InstructionWrapper operator!() const;
InstructionWrapper operator*(const InstructionWrapper &other) const;
InstructionWrapper operator/(const InstructionWrapper &other) const;
InstructionWrapper operator%(const InstructionWrapper &other) const;
InstructionWrapper operator<(const InstructionWrapper &other) const;
InstructionWrapper operator>(const InstructionWrapper &other) const;
InstructionWrapper operator>=(const InstructionWrapper &other) const;
InstructionWrapper operator<=(const InstructionWrapper &other) const;
InstructionWrapper operator==(const InstructionWrapper &other) const;
InstructionWrapper operator!=(const InstructionWrapper &other) const;
InstructionWrapper operator++();
InstructionWrapper operator++(int);
InstructionWrapper operator--();
InstructionWrapper operator--(int);
```

```
InstructionWrapper field(const cecko::CIName &field_name,
    cecko::CKTypeSafeObs field_type) const;
InstructionWrapper field(unsigned int field_id, cecko::CKTypeSafeObs
    field_type) const;

/// @brief generates instruction representing bool value, which
/// says wether this Tagged type pointer has given tag value
InstructionWrapper has_tag(int tag) const;

// cast functions
InstructionWrapper to_int() const;
InstructionWrapper to_char() const;
InstructionWrapper to_ptr() const;
InstructionWrapper to_num() const;
InstructionWrapper to_bool() const;
InstructionWrapper to_type(cecko::CKTypeSafeObs to_type) const;
InstructionWrapper to_tagged(cecko::CKTypeSafeObs tagged_type) const;
InstructionWrapper to_tagged(const std::string &tagged_type_label) const;

static std::string get_struct_type_name(cecko::CKTypeSafeObs t);
static InstructionWrapper null_inst(cecko::context *ctx,
    cecko::CKPtrTypeSafeObs expected_type);

};
```

**Figure A.1** An instruction wrapper class used to represent general run-time expressions and operations on them

## A.2 StaFip grammar

```
translation_unit:
    external_declaration
    | translation_unit external_declaration

external_declaration:
    function_definition
    | enumtype_decl_specifier

new_lines:
    NEWLINE
    | new_lines NEWLINE

expression_end:
    new_lines
    | EOF

function_definition:
    function_definition_head expression
    | function_definition_info ; expression_end

function_definition_info:
    declaration_specifiers declarator

function_definition_head:
    function_definition_info ASGN

primary_expression:
    INTLIT
    | STRLIT
    | ( expression_body )

postfix_expression:
    primary_expression
    | IDF ( argument_expression_list )
    | IDF

argument_expression_list:
    assignment_expression
    | argument_expression_list , assignment_expression
    | %empty
```

```
unary_expression:
    postfix_expression
    | unary_operator cast_expression

unary_operator:
    &
    | *
    | ADDOP
    | !

cast_expression:
    unary_expression

multiplicative_expression:
    cast_expression
    | multiplicative_expression * cast_expression
    | multiplicative_expression DIVOP cast_expression

additive_expression:
    multiplicative_expression
    | additive_expression ADDOP multiplicative_expression

relational_expression:
    additive_expression
    | relational_expression CMPO additive_expression

equality_expression:
    relational_expression
    | equality_expression CMPE relational_expression

logical_and_expression:
    equality_expression
    | logical_and_expression && equality_expression

logical_or_expression:
    logical_and_expression
    | logical_or_expression || logical_and_expression


assignment_expression:
    logical_or_expression
    | unary_expression assignment_operator assignment_expression

assignment_operator:
    =
```

```
match_head:
    MATCH assignment_expression -> declaration_specifiers

match_expression:
    assignment_expression
    | match_binders_list block_end

match_binders_list:
    match_binders_list_head_start expression_body
    | match_binders_list_head expression_body

match_binders_list_head_start:
    match_head block_start match_binder_head

match_binders_list_head:
    match_binders_list match_binder_head

match_binder_head:
    | match_binder_definer ->

match_binder_definer:
    IDF ( match_binder_arguments_list )
    | IDF

match_binder_arguments_list:
    IDF
    | match_binder_arguments_list , IDF

expression_body:
    match_expression
    | flow_expression

expression:
    expression_body expression_end

declaration_specifiers:
    declaration_specifier
    | declaration_specifier declaration_specifiers

declaration_specifier:
    type_specifier_qualifier
    | FIP

type_specifier:
    ETYPE
    | declared_type_name
    | typedef_name
```

```
declared_type_name:
    TYPEDEF IDF

enumtype_decl_head:
    TYPEDEF IDF

block_start:
    {
    | { new_lines

block_end:
    }
    | new_lines }

enumtype_decl_specifier:
    enumtype_decl_head block_start member_types_declaration_list
                                                block_end new_lines

member_types_declaration_list:
    member_types_declaration
    | member_types_declaration_list member_types_declaration


member_types_declaration:
    IDF ( member_declaration_list ) ;
    | IDF ;

member_declaration_list:
    member_declaration
    | member_declaration_list , member_declaration

member_declaration:
    specifier_qualifier_list member_declarator


specifier_qualifier_list:
    type_specifier_qualifier
    | type_specifier_qualifier specifier_qualifier_list

type_specifier_qualifier:
    type_specifier

member_declarator:
    declarator
```

```
declarator:
    pointer direct_declarator
    | direct_declarator

pointer:
    * pointer
    | *

direct_declarator:
    IDF
    | function_declarator

function_declarator:
    direct_declarator [ parameter_type_list ]

parameter_type_list:
    parameter_list
    | %empty

parameter_list:
    parameter_declaration
    | parameter_list , parameter_declaration

parameter_declaration:
    declaration_specifiers declarator
    | declaration_specifiers

typedef_name:
    TYPEIDF


expression_statement:
    match_expression ;


if_expression_head:
    IF ( expression_body )

if_non_split_expression:
    if_expression_head non_split_expression
```

```
if_non_split_expression_else:
    if_non_split_expression ELSE

flow_expression:
    non_split_expression
    | split_expression

non_split_expression:
    if_non_split_expression_else non_split_expression
    | expression_statement

split_expression:
     if_expression_head expression_body
     | if_non_split_expression_else split_expression
```

**Figure A.2** The StaFip Grammer where translation_unit is the root nonterminal (%start)

## A.3 Attachment Archive Structure

The following listing shows the structure of the archive containing the source code of the project.

```
FipCompiler/
├─ CMakeLists.txt
├─ README.md ...        Contains information about the project.
├─ generate_fip_tests_data.sh ...    Builds the project, runs
│                                    the benchmarks.
├─ verify_stafip.sh ...     Builds the project and runs
│                           the verification tests.
├─ CMakeConf.cmake
├─ DebugLog.txt ...     Used for logging compilation using the
│                       StaFip compiler.
├─ .github/
│  └─ workflows/
│     └─ verify.yaml ...    Define the CI action with
│                           verification tests.
├─ banchmarks_utils/
│  ├─ compile_stafip_scripts_in.sh ...    In a given folder,
│  │                                      compile all .ffip
│  │                                      files using the StaFip
│  │                                      compiler.
│  ├─ generate_banchmarks_data.sh ...    Takes care of the figure
│  │                                     generation of the results.
│  └─ generate_graphs.py ...    Takes care of the graph
│                               generation of the results.
├─ bench/ ...    Benchmark scripts run by the
│  │             generate_fip_tests_data.sh script.
│  ├─ finger_trees_fip.ffip
│  ├─ finger_trees_normal.ffip
│  ├─ quick_sort_fip.ffip
│  ├─ quick_sort_normal.ffip
│  ├─ red_black_trees_fip.ffip
│  └─ red_black_trees_normal.ffip
├─ compiled_programs_data/
│  └─ .lib/ ...    The C library linked to the .ffip executables.
│     ├─ ck_utils.cpp
│     └─ ck_utils.hpp
└─ doc/ ...    Contains the project documentation.
   ├─ StaFip_specification.md
   └─ StasFip_stdlib_docs.md
```

```
│
│  │
│  ├── developer_docs.md
│  └── doxyfile
├── fmwk/ ...      The Cecko library folder.  Abstracts the LLVM library.
│  │
│  └── ...
├── main/
│  └── ...
├── solution/ ...      Holds the implementation of the StaFip compiler.
│  ├── CMakeLists.txt
│  ├── calexer.lex
│  ├── caparser.y
│  ├── casem.cpp
│  └── casem.hpp
└── test/ ...      The verification test scripts and expected
   │               results in the .gold files.
   ├── basic_list_travers.ffip
   ├── basic_no_code.ffip
   ├── finger_trees_fip.ffip
   ├── quick_sort_normal.ffip
   ├── basic_list_travers.ffip.gold
   ├── basic_no_code.ffip.gold
   ├── finger_trees_fip.ffip.gold
   ├── quick_sort_normal.ffip.gold
   ├── basic_if.ffip
   ├── basic_match.ffip
   ├── basic_null_type_match.ffip
   ├── finger_trees_normal.ffip
   ├── red_black_trees_fip.ffip
   ├── basic_if.ffip.gold
   ├── basic_match.ffip.gold
   ├── basic_null_type_match.ffip.gold
   ├── finger_trees_normal.ffip.gold
   ├── red_black_trees_fip.ffip.gold
   ├── basic_list_reverse.ffip
   ├── basic_match_if_match.ffip
   ├── basic_types.ffip
   ├── quick_sort_fip.ffip
   ├── red_black_trees_normal.ffip
   ├── basic_list_reverse.ffip.gold
   ├── basic_match_if_match.ffip.gold
   ├── basic_types.ffip.gold
   ├── quick_sort_fip.ffip.gold
   └── red_black_trees_normal.ffip.gold
```

**CAPTION: The Black entries are the work of this thesis while the Gray are part of the Cecko Skeleton [7]. The project is also available on GitHub: `https://github.com/JaromirProchazka/FipCompiler`.**