



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

---

# **GOOPEA: A Functional Programming Language with FIP Functionality**

Bachelor's thesis in Computer science and engineering

OSCAR CARLSSON  
OLIVER IVARSSON  
ELLA LINDSTRÖM  
GUSTAV NORÉN  
PATRIK WAHLGREN  
ALEXANDER WAHLSTEN

---

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2025



BACHELOR'S THESIS 2025

# GOOPEA: A Functional Programming Language with FIP Functionality

OSCAR CARLSSON  
OLIVER IVARSSON  
ELLA LINDSTRÖM  
GUSTAV NORÉN  
PATRIK WAHLGREN  
ALEXANDER WAHLSTEN



UNIVERSITY OF  
GOTHENBURG

---



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2025

GOOPEA: A Functional Programming Language with FIP Functionality

OSCAR CARLSSON OLIVER IVARSSON ELLA LINDSTRÖM GUSTAV NORÉN  
PATRIK WAHLGREN ALEXANDER WAHLSTEN

© OSCAR CARLSSON, OLIVER IVARSSON, ELLA LINDSTRÖM, GUSTAV NORÉN,  
PATRIK WAHLGREN, ALEXANDER WAHLSTEN 2025.

Supervisor: Magnus Myreen, Department of Computer Science and Engineering  
Examiners: Patrik Jansson and Arne Linde, Department of Computer Science and  
Engineering  
Graded by teacher: Wolfgang Ahrendt, Department of Computer Science and En-  
gineering

Bachelor's Thesis 2025  
Department of Computer Science and Engineering  
Chalmers University of Technology and University of Gothenburg  
SE-412 96 Gothenburg  
Telephone +46 31 772 1000

Typeset in L<sup>A</sup>T<sub>E</sub>X  
Gothenburg, Sweden 2025

# GOOPEA: A Functional Programming Language with FIP Functionality

OSCAR CARLSSON, OLIVER IVARSSON, ELLA LINDSTRÖM, GUSTAV NORÉN,  
PATRIK WAHLGREN, ALEXANDER WAHLSTEN

Department of Computer Science and Engineering  
Chalmers University of Technology and University of Gothenburg

## Abstract

Functional programming languages offer a unique way of writing code which can often be very concise, elegant and intuitive. However, due to their high-level nature, they are often more memory inefficient than other languages. This is further exacerbated by the memory management strategy *reference counting* which requires extra memory.

This project explores a solution to this problem by extending reference counting with Functional In-Place (FIP) optimization in a new programming language GOOPEA. The core idea of FIP is that objects that are referenced precisely once are reused.

A web playground to experiment with GOOPEA was also developed. The playground supports writing and executing of GOOPEA code, contains documentation, and can show a visualization of the state of the memory for every step of the program.

We ran benchmarks of multiple programs, comparing the performance with and without FIP. The tests showed that FIP greatly reduced memory usage in most programs, and got a significant speed improvement from the reduced number of memory allocations.

---

## Sammandrag

Funktionella programmeringsspråk erbjuder ett unikt sätt att skriva kod som ofta är koncist, elegant och intuitivt. Eftersom de är högnivåspråk är de dock ofta mer ineffektiva när det gäller minneshantering än andra språk. Detta förvärras av minneshanteringsstrategin *reference counting* som kräver extra minne.

Detta projekt utforskar en lösning till detta problem genom att bygga ut på *reference counting* med Functional In-Place (FIP) optimering i ett nytt programmeringsspråk GOOPEA. Idén bakom FIP är att objekt som är refererade till exakt en gång återanvänds.

För att experimentera med GOOPEA så utvecklades också en webbplayground. Webbplaygrounden stödjer skrivningen och exekveringen av GOOPEA-kod, innehåller dokumentation och kan visa en visualisering av minnet efter varje steg av programmet.

Vi körde prestandatest av flera program, och jämförde prestandan med och utan FIP. Testerna visade att FIP minskade minnesanvändning i de flesta program, och gav en betydlig hastighetsförbättring tack vare det minskade antalet minnesallokeringar.

# Acknowledgements

We would like to sincerely thank our supervisor, Magnus Myreen, for the insight and guidance he provided us. It was at his suggestion that we decided to explore the benefits of FIP, and we all feel a sense of gratitude for him being there for us, from the start of this thesis to its completion.

Oscar Carlsson, Oliver Ivarsson, Ella Lindström, Gustav Norén, Patrik Wahlgren,  
Alexander Wahlsten, Gothenburg, June 2025



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Purpose . . . . .	2
1.2	Design Scope . . . . .	2
1.3	Pipeline of Compilation . . . . .	3
1.4	Structure of report . . . . .	4
<b>2</b>	<b>Creating a Functional Programming Language</b>	<b>5</b>
2.1	An Overview of GOOPEA . . . . .	5
2.1.1	Basic syntax . . . . .	5
2.1.2	Data Types . . . . .	6
2.1.3	Traversing a List . . . . .	6
2.1.4	Pure functions . . . . .	7
2.1.5	Let Bindings and Unboxed Tuples . . . . .	7
2.1.6	Reversing a list . . . . .	8
2.2	Lexing the source code into a token stream . . . . .	9
2.3	Parsing the token stream with a context-free grammar . . . . .	10
2.3.1	Context-free Grammar . . . . .	10
2.3.2	Using LALRPOP as a parser generator . . . . .	11
2.4	Semantic Analysis . . . . .	13
2.5	The Code Generation . . . . .	13
2.5.1	A Sequentially Transformed Intermediate Representation . . . . .	14
2.5.2	Memory Management . . . . .	16
2.5.3	Garbage Collection . . . . .	16
2.5.4	Reference Counting . . . . .	17
2.5.5	Adding scope-based Reference Counting to our IR . . . . .	19
2.5.6	Core & GOOPEA's Runtime Representations . . . . .	20
<b>3</b>	<b>An Interesting Idea: FIP</b>	<b>23</b>
3.1	Extending the IR with Reuse . . . . .	23
3.2	Borrowing . . . . .	26
3.3	More Precise Reference Counting . . . . .	28
<b>4</b>	<b>An Interpreter for GOOPEA</b>	<b>31</b>
4.1	Overview . . . . .	31
4.2	Runtime Architecture . . . . .	31
4.3	Execution Model . . . . .	32

4.3.1	Expression Evaluation . . . . .	32
4.3.2	Function Calls and Scoping . . . . .	32
4.3.3	Control Flow . . . . .	33
4.4	Memory Management and FIP . . . . .	33
4.5	Debugging and Web Integration . . . . .	33
4.6	Performance and Benchmarking . . . . .	34
<b>5</b>	<b>Putting Everything Together in a Web Playground</b>	<b>35</b>
5.1	The Editor Page . . . . .	35
5.1.1	Output . . . . .	37
5.1.2	Debugging . . . . .	37
5.1.3	Compilation . . . . .	37
5.1.4	Visualization . . . . .	39
5.2	The Example Page . . . . .	40
5.3	The Documentation Page . . . . .	41
5.4	Common code and shared classes . . . . .	41
5.4.1	The Navigation Bar . . . . .	42
5.5	Summary of the playground’s features . . . . .	43
<b>6</b>	<b>Result, Discussion, and Conclusion</b>	<b>45</b>
6.1	Future Work . . . . .	46
6.2	Societal and Ethical Aspects . . . . .	47
<b>A</b>	<b>Benchmarks</b>	<b>51</b>

# Abbreviations

**FIP** Functional In-Place

**ADT** Algebraic Data Type

**AST** Abstract Syntax Tree

**RC** Reference Counting

**JS** JavaScript

**IR** Intermediate Representation

# 1

## Introduction

Modern software demands code that is maintainable, easy to write, and free from bugs. With an emphasis on purity and often accompanied by strong static type-systems, the functional style attracts many programmers as an option. Functional programming has its roots in lambda-calculus and model computations by function composition, as opposed to traditional imperative languages that model it by a sequence of instructions to execute. This makes it particularly elegant for modeling recursive relationships, requiring less code than the alternatives.

However, at the lowest level, computers are only able to execute imperative machine code. Since this is an entirely different paradigm, compiled functional code may often run slower than code from an imperative counterpart. Its high level nature also often makes it hard or impossible to implement certain classes of optimizations that other languages can. For these reasons, functional languages are often uncondusive for projects where performance is critical. This is where a technique called Functional In-Place, abbreviated as FIP, emerges as a way to alleviate this problem.

FIP is an optimization to reuse memory and avoid unnecessary (de)allocations. To start, take a look at the code in Code 1. It is a program written in our language GOOPEA, and it contains an algebraic data type (ADT) `List` and a function `increment_list` that increments every element of a list by one. As can be seen at line 1, a `List` can either be empty (`Nil`), or it can be a node that contains an element and a reference to the next `List` instance (`Cons`). This makes `List` a linked list, and it will be allocated on the heap.

```
1 enum List = Nil, Cons(Int, List);
2
3 fip List: List
4 increment_list list = match list {
5   Nil: Nil,
6   Cons(x, rest): Cons(x + 1, increment_list rest)
7 };
```

*Code 1: Function that increments a list in GOOPEA. Note the function signature definition at line 3. The `fip` keyword makes the compiler compile with FIP optimization, and the `rest` indicates that the function returns a `List` and takes a `List` as an argument.*

Notice how, at line 5 and 6 a new allocation is made every time `increment_list` is called. When the function is finished, an entirely new list will have been constructed,

doubling the original memory. However, if the original list is not referenced or used anywhere else in the program once the function completes, it will eventually simply get deallocated. This is an inefficiency, as the original memory could have simply been edited and reused without any effect on the program. This is the core idea of FIP: identify where memory can be reused, and reuse it.

### 1.1 Purpose

The purpose of this project is to develop a functional programming language that supports both FIP and non-FIP functions, as well as a web playground for the language. The goal is to make FIP functionality more freely available and easy to experiment with. The project measures the differences in performance between FIP and non-FIP functions to determine their benefit in a general-purpose functional programming language. Ultimately, it answers the question: are FIP functions desirable when programming?

### 1.2 Design Scope

With these goals in mind, we decided to design GOOPEA as a minimal functional programming language tailored for experimenting with FIP. To support this, we selected a small set of features balancing functionality with simplicity:

- Named top-level functions
- Unboxed tuples
- `|let <var> = ... in ...|` expressions
- Integers and integer arithmetic
- Algebraic Data Types (ADT)
- Pattern matching
- Reference Counting
- Toggleable FIP optimization

The most fundamental feature of a functional programming language is the ability to define and call functions. We decided that these should be named, and only definable in the topmost level of the program, i.e no anonymous nor nested functions. They are also pure, always returning the same output for the same input. We additionally decided that each function should be able to have an arbitrary number of arguments and return-values via unboxed tuples. `let in` expressions are then able to be bind into unboxed tuples, but also allow for neatly storing intermediate results in other parts of the code.

To create different types of data we decided to include ADTs and integers. To manipulate these we naturally also included basic integer operations, as well as pattern matching through `match` statements. These are able to destruct ADTs and change control flow.

Lastly we chose the memory management strategy as Reference Counting. This is

then conditionally able to be further optimized with FIP by writing the `fip` keyword in the beginning of a function.

To support and demonstrate these features, we decided to build a web playground that would allow the user to write and run code, view the output and Intermediate Representations (IRs) of said code, see example code snippets, and read documentation. To run the code on the web, we also decided to create an interpreter that could interpret the C-like IR produced by the compiler.

We also made deliberate choices to exclude certain functionalities to maintain simplicity and focus on FIP. Common language features we did not implement are:

- Primitive data types other than integers
- Higher order functions
- Polymorphism

These exclusions allowed us to use a simpler type system, while still being able to reach our goal of testing FIP.

### 1.3 Pipeline of Compilation

GOOPEA code is written in the front-end web playground and is then sent through several stages. The lexer, parser, and semantic analyzer checks for faults with the code, for example type errors and syntax errors. The code generator is then responsible for translating the processed GOOPEA code into a C-like intermediate representation (IR). This IR is either able to be synthesized into actual C code or executed by the interpreter on the web playground. The resulting data from the interpreter (memory allocations, variables, call stack, etc.) is finally given back to the web playground for the users perusal. The pipeline described above is shown in Fig. 1.1.

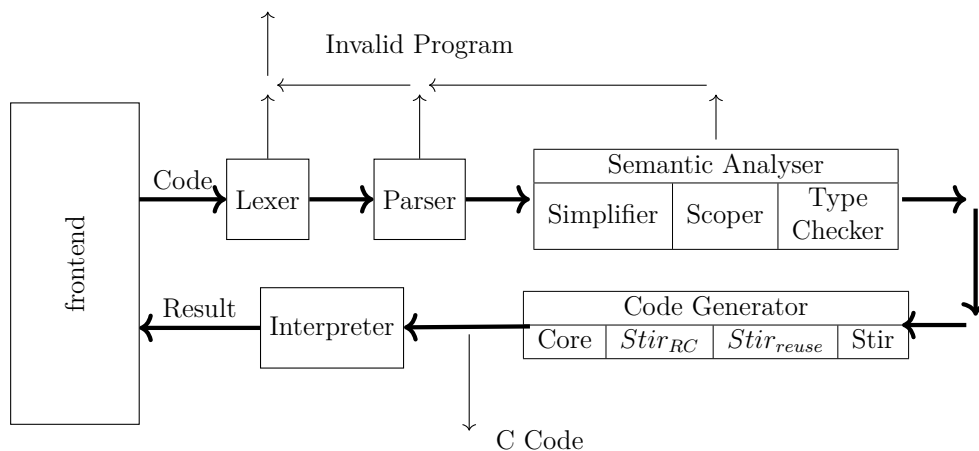


Figure 1.1: The complete pipeline of compiling GOOPEA. The various steps are covered in chronological order throughout the next sections of this report.

## 1.4 Structure of report

The rest of the report is structured as follows: Section 2 describes the implementation of GOOPEA as a functional programming language, including the lexer, parser, semantic analyzer, and the code generation process. Section 3 covers large part of the code generation and adding FIP to GOOPEA. Section 4 describes the process of interpreting C and section 5 the web playground. Each of these sections will contain first the background and theory of their main idea and then our method for implementing it to create our language.

# 2

## Creating a Functional Programming Language

In this section, we present how we created a functional programming language. Background and methodology are interwoven in this section and section 3. We first present the necessary background and then describe our implementation for the given theory.

### 2.1 An Overview of GOOPEA

Functional programming is a major programming paradigm along with object-oriented programming and imperative programming. A typical functional program obtains its desired result by calling functions and compositions of several functions in a chain. An overview of functional programming and how it is applied will be shown with GOOPEA.

#### 2.1.1 Basic syntax

```
1 Int : Int                1 plustwo :: Int -> Int
2 plustwo x = x + 2;      2 plustwo x = x + 2
```

*Code 2: A function that adds two to its argument, written in GOOPEA (left) and Haskell (right).*

```
1 Bool : Bool              1 not :: Bool -> Bool
2 not b = match b {       2 not b = case b of
3   True : False,        3   True -> False
4   False : True         4   False -> True
5 };
```

*Code 3: A function that negates its argument, written in GOOPEA (left) and Haskell (right). `match b` or `case b` of checks what value `b` is. If it's `True` then it evaluates to `False` and if `False` then `True`.*

### 2.1.2 Data Types

GOOPEA and other functional languages use ADTs to create new data types. In fact, `Bool` in GOOPEA is defined as an ADT, which can be seen in Code 4.

```
1 enum Bool = True, False;
```

*Code 4: ADT of a boolean defined in GOOPEA.*

This is similar to how enums in other languages work in that the only two values that a Boolean may have are `True` or `False`. However, unlike regular enums, ADTs let us define complex data types concisely with constructors. Constructors let one create data with values of specified types or chain types of itself which can mimic the behavior of linked lists or trees. A declaration of a linked list of integers is given in Code 5, and an example of a list definition is given in Code 6.

```
1 enum IntList = Nil, Cons(Int, IntList);
```

*Code 5: ADT construction of a list of integers in GOOPEA. The name of the constructor here is “Cons”.*

```
1 () : IntList
2 xs = Cons(3, Cons(2, Cons(1, Nil)));
3 () : IntList
4 ys = Cons(1, xs);
```

*Code 6: `xs` is declared as defined in Code 5. In order to end the chain of recursively adding lists, `Nil` is added instead. Note how `xs` is used as an argument to define a list in `ys`.*

### 2.1.3 Traversing a List

GOOPEA does not have mutable state, which means that one cannot run a program through a for-loop. Instead, GOOPEA depends on recursion to process lists. The following example in Code 7 shows how the recursive declaration of `length` is used to obtain the length of the list.

```
1 enum IntList = Nil, Cons(Int, IntList);
2
3 IntList : Int
4 length xs = match xs {
5     Nil : 0,
6     Cons(x, ys) : 1 + length ys
7 };
```

*Code 7: Function to get the length of a list in GOOPEA.*

`length` takes in an `IntList` and returns an `Int`. It computes this by checking the cases of what `xs` can be. This is called pattern matching. If the list is `Nil` then its length is 0, and if it isn't then we know it has to have at least one element. By recursing through the list, we can account for all elements and get the length of the list. This is shown in Table 2.1. The calculated values are summed on the way out of the recursive calls.

Table 2.1: Step-by-step evaluation of the `length` function on `Cons(4, Cons(1, Cons(6, Nil)))`, as defined in Code 7.

Step	Expression
0	<code>length (Cons(4, Cons(1, Cons(6, Nil))))</code>
1	<code>1 + length (Cons(1, Cons(6, Nil)))</code>
2	<code>1 + 1 + length (Cons(6, Nil))</code>
3	<code>1 + 1 + 1 + length Nil</code>
4	<code>1 + 1 + 1 + 0</code>
5	<code>3</code>

### 2.1.4 Pure functions

GOOPEA implements pure functions, which means that the same input *always* evaluates to the same output and there are no side effects (e.g. a variable being modified).

### 2.1.5 Let Bindings and Unboxed Tuples

The `let` keyword allows the result of a computation to be bound to a variable that can be used in later expressions. An example usage of the syntax is given in Code 8.

```
1 Int : Int
2 fun x = let y = plustwo x in y * 2;
```

Code 8: Binds the result of `plustwo` defined in Code 2 to `y`, which then uses the result of `y` and multiplies it by 2.

The pattern also allows one to bind unboxed tuples, as can be seen in Code 9

```
1 IntList: (IntList, IntList)
2 split xs = match xs {
3   Nil: (Nil, Nil),
4   Cons(x, xx): match xx {
5     Nil: (Cons(x, Nil), Nil),
6     Cons(y, yy): let (left, right) = split yy in
7       (Cons(x, left), Cons(y, right))
8   }
9 };
```

Code 9: The function takes a list and splits it into two. The result of `split yy` gets bound to `left` and `right` and are used for the end result in the list constructions after the keyword `in`.

### 2.1.6 Reversing a list

A final example is shown in Code 10.

```
1 enum IntList = Nil, Cons(Int, IntList);
2
3 IntList : IntList
4 reverseList xs = reverseHelper(xs, Nil);
5
6 (IntList, IntList) : IntList
7 reverseHelper(xs, acc) = match xs {
8   Nil : acc,
9   Cons(x, ys) : reverseHelper(ys, Cons(x, acc))
10 };
```

Code 10: GOOPEA code to reverse a list in  $O(n)$ .

Lists are “first in last out”, and as such we get a natural way to reverse a list by taking the integer from one list and appending it to another. `reverseHelper` keeps track of the two lists through the two parameters. The function then returns the reversed list when the original list argument no longer has any elements. The steps of iteration are illustrated in Table 2.2.

Table 2.2: Step-by-step iterations of how the parameters `xs` and `acc` in `reverseHelper` changes over time.

Iter	<code>xs</code>	<code>acc</code>
0	<code>Cons(1, Cons(2, Cons(3, Nil)))</code>	<code>Nil</code>
1	<code>Cons(2, Cons(3, Nil))</code>	<code>Cons(1, Nil)</code>
2	<code>Cons(3, Nil)</code>	<code>Cons(2, Cons(1, Nil))</code>
3	<code>Nil</code>	<code>Cons(3, Cons(2, Cons(1, Nil)))</code>

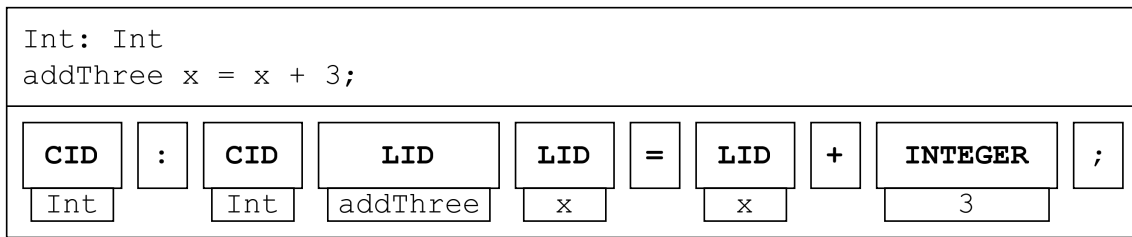


Figure 2.1: Shows how a program containing the function `addThree` gets tokenized. The program is in the top cell and its token stream is in the bottom cell. Note how `addThree` and `x` both get lexed as `LID`, even though their text differs. Also note that this text is stored within the token.

## 2.2 Lexing the source code into a token stream

Lexing is the first step in order to compile and eventually execute code written in GOOPEA. This is done by splitting the code into a stream of tokens. Each token represents some type of syntactic element of the code, for example an identifier, operator, numeric literal, or delimiter. The following is a list of all possible tokens in GOOPEA:

- List of symbols: ( ) { } : ; , = \_ + - \* / % < <= == >= >
- List of keywords:
  - FIP: `'fip'`
  - MATCH: `'match'`
  - LET: `'let'`
  - IN: `'in'`
- INTEGER: Non-negative integer literal, i.e '5', '10', '0304'...
- LID: Identifier starting with lowercase letter
- CID: Identifier starting with capital letter

Fig. 2.1 shows an example of how a program is tokenized. Notice how all types of whitespace gets discarded, as well as how identifiers starting with a lowercase letter and those with a uppercase letter gets lexed as different tokens: `LID` and `CID` respectively. This is because `LID` exclusively refer to variables and functions, while `CID` exclusively refer to ADTs and constructors. Whether any given `LID` refers to a variable or a function is unknowable at this stage without more context, and will be resolved later.

The lexer itself is implemented using the library `logos` [12], which is a fast and easy to use lexer. Each type of token is defined via a regular expression and an optional priority. Any code with a token that doesn't match the defined regex thus produces an error. `Logos` also provides each token with data about its location within the source code.

## 2.3 Parsing the token stream with a context-free grammar

The next step of the compilation process is to turn the linear token stream into an intermediate representation (shortened as IR) tree using a parser, which captures all structural aspects of the code. The parser that does this is generated via the library LALRPOP [14] through a context-free grammar defined for the language.

### 2.3.1 Context-free Grammar

Similarly to spoken languages, context-free grammars for programming languages are made up of a set of words (terminals), and some grammatical rules (productions) [13]. For this project, the set of terminals, denoted by  $\Sigma$ , is the same as the set of tokens produced by the lexer. To explain the semantics behind a context-free grammar, we will use a subset of the full grammar where  $\Sigma = \{\text{INTEGER}, \text{LID}, (, ), +, -, *, /\}$  and a couple of productions:

Expr	::=	LID	Variable
			INTEGER
			Expr Op Expr
			( Expr )
Op	::=	+   -   *   /	Arithmetic operators

The set  $V = \{\text{Expr}, \text{Op}\}$  is called the set of non-terminals, also known as variables, and Expr is the starting symbol. A program is in the language of the context-free grammar, i.e. it is syntactically valid, if there exists a sequence of steps that transforms the starting symbol into said program. Each step consists of selecting a non-terminal in the current string, choosing one of the productions from this non-terminal, and then substituting the non-terminal in the string with the chosen production's right-hand side. The following is an example of how the program  $5 * (x + 3)$ , which when lexed becomes `INTEGER * (LID + INTEGER)`, might be derived:

$$\text{Expr} \implies \text{Expr} + \text{Expr} \tag{2.1}$$

$$\implies \text{Expr} + (\text{Expr}) \tag{2.2}$$

$$\implies \text{Expr} * (\text{Expr} + \text{Expr}) \tag{2.3}$$

$$\implies \text{INTEGER} * (\text{Expr} + \text{Expr}) \tag{2.4}$$

$$\implies \text{INTEGER} * (\text{LID} + \text{Expr}) \tag{2.5}$$

$$\implies \text{INTEGER} * (\text{LID} + \text{INTEGER}) \tag{2.6}$$

Thus, we can see that this is a valid program, and more generally that this grammar defines a language of arithmetic expressions over integers and variables. From a derivation such as this one it is also possible, and often very useful, to construct a *parse tree*. Fig. 2.2 shows hierarchically how the derivation was constructed and forms the basis for constructing the IR, which will be further explained in a later section.

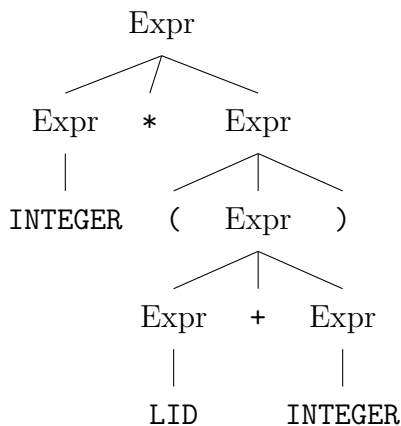


Figure 2.2: Parse tree from the derivation in Equation 2.6

### 2.3.2 Using lalrpop as a parser generator

A parser is, simply put, a program that consumes a token-stream and produces either a parse tree or a syntax error. In GOOPEA, the parser is generated from a grammar file via the Rust library LALRPOP [14]. LALRPOP is a fast and convenient parser generator and supports some special syntax that makes writing the grammars easier. These are the ? operator, the \* operator, and macros. These don't make the grammar more powerful, i.e. they don't allow the grammar to represent a wider class of languages, but simply increase readability and reduce common parts. To explain their meaning, take the following grammar:

```

A           ::= FIP?
B           ::= LID*
CommaList<X> ::= X
              | X , CommaList<X>
C           ::= CommaList<CID>
  
```

The ? operator means 0 or 1 occurrences of the preceding symbol, so from A we can either derive | or |FIP|. Similarly, the operator \* means 0 or more occurrences of the preceding symbol, so from B we can derive |, |LID|, |LID LID|, etc. Lastly, CommaList<X> is an example of a macro, and they can be viewed as a production template. X here is neither a terminal nor a non-terminal, but simply a stand-in symbol that can be instantiated with any terminal or non-terminal. Therefore, C can be derived into |CID|, |CID , CID|, |CID , CID , CID|, etc.

Another important note about LALRPOP is that it generates an LR(1) parser from a context-free LR(1) grammar. While a complete explanation of what LR(1) means is out of the scope of this report, essentially it means that the grammar has to be written in a way where it is unambiguous and 'simple to parse'<sup>1</sup>. A grammar is called ambiguous if it is possible to generate two different parse trees for the same derivation.

<sup>1</sup>The parser must be able to decide what action to take for the current symbol while only looking at the next terminal in the string. This is where the 1 in LR(1) comes from.

The grammar in Fig. 2.3 shows the essence of the grammar used for GOOPEA. The actual grammar is written slightly differently, but this is more readable as an overview. The main simplification in this presentation is regarding the `Tuple<X>` production, as in the full grammar it is, for example, allowed to skip parenthesis if there is 1 or 0 arguments, i.e `inc(3)` could be written as `inc 3`, and `pi()` as simply `pi`. The grammar file also contains Rust code snippets for each production to create the IR, which make it so that the IR can be much more concise than a simple parse tree.

Program	::=	Definition*	
Definition	::=	ENUM CID = CommaList<Cons> ;	Enum
		Sig LID Tuple<LID> = Expr ;	Func
Sig	::=	FIP? Tuple<Type> : Tuple<Type>	Signature
Cons	::=	CID Tuple<Type>	
Type	::=	CID	
Expr	::=	LID	Var/Func
		LID Expr	Func
		CID	Constructor
		CID Expr	Constructor
		INTEGER	Literal
		Tuple<Expr>	Tuple
		MATCH Expr { CommaList<Case> }	
		LET Tuple<LID> = Expr IN Expr	
		+ Expr	
		- Expr	
		Expr Op Exp	
Op	::=	+   -   *   /   %   ==   !=   <   <=   >   =>	
Case	::=	Pattern : Expr	
Pattern	::=	CID Tuple<WildVar>	Constructor
		INTEGER	
		LID	Variable
WildVar	::=	CID	Variable
		-	Wildcard
CommaList<X>	::=	X	
		X , CommaList<X>	
Tuple<X>	::=	( CommaList<X> )	

Figure 2.3: A slightly modified version of the grammar used for GOOPEA. Notice how there is no production for simply wrapping an expression in parenthesis. This is instead implemented via the `Tuple` production.

## 2.4 Semantic Analysis

Semantic analysis is the final stage of analysis of the compiler, and thus the last stage invalid programs may reach. During semantic analysis the IR will get transformed and gain additional data multiple times. This data is then used for further transformations and during even later stages of compilation. The three main transformation are simplification, variable scoping, and type inference.

The simplification transformation aims to reduce the complexity of the parse tree in two ways. Firstly, it reclassifies operations as function calls, with the operator becoming the function name. This means that `1+6` becomes `+(1,6)`, which greatly reduces code duplication. Secondly, the simplification transformation makes it so that all match statements only match on variables. This is done by substituting expressions of the form `match <expr> { ... }` with `let _ = <expr> in match _ { ... }`. Note that underscore is the variable name, and because it normally isn't allowed as an identifier name, introducing it here will not have any impact on the rest of the code.

The variable scoping transformation resolves what different identifiers refer to, and gives each expression node data about its scope. This data is modeled as a map from variable name to new unique ids, resolving ambiguities regarding shadowing. It is also this step that differentiates identifiers starting with lowercase as either variables or functions.

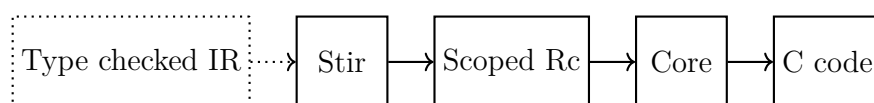
The final transformation, type inference, infers for each expression node its type and rejects programs with inconsistent typing. A type is either an integer, one of the ADTs defined in the program, or an unboxed tuple consisting of other types. The unboxed tuple type is special and, due to the nature of unboxed tuples, is only allowed in places where it is immediately returned, bound via a `let` expression, or passed into a function. The type inference itself is implemented recursively.

## 2.5 The Code Generation

Code generation is the compilation phase where the IR gets translated into a lower level target IR. The most important requirement in this translation is that the target code must preserve the semantic meaning of the source code while simultaneously producing correct Code [1, Ch. 8].

It is seldom the case for programming languages that source code is compiled directly to target code. Instead, the IR goes through several stages where the abstraction level gradually decreases between each pass. It is also in these stages where we analyze the code and perform optimizations [1, Ch. 8].

The code generation in GOOPEA consists of four stages, as can be seen in Fig. 2.4.



*Figure 2.4: The pipeline of code generation*

*Stir* is a transformed representation which simplifies the IR for later optimizations. In the *Rc* stage we add a strategy for memory management to the IR. The *Core* stage then prepares the IR to be generated into C code. Finally, the IR is translated into C code.

### 2.5.1 A Sequentially Transformed Intermediate Representation

Historically, a popular choice of intermediate representation when compiling functional languages was using CPS-style [2]. However, in 1993, Flanagan et al. showed that the total effects of the three phases in CPS is equivalent to a source-to-source ANF-transformation that simulates the compaction phase of CPS[8]. Administrative normal form (ANF), as described in Flanagan et al, is an algorithm performed on a core scheme which is a representation of a functional language. The main property of the scheme is that each non-atomic expression (an expression that is not bound to some variable) must be bound to some variable before it can be used. Let us provide an example in terms of GOOPEA code to build a deeper understanding.

```
1 Int : List
2 build n = match n == 0 {
3   True: Nil,
4   False: Cons(n, build(n - 1))
5 };
```

*Code 11: GOOPEA function for creating a list with n elements*

The code in Code 11 is *not* in ANF form. The first problem lies in the match expression `n == 0`. Here the value zero is used without binding it first<sup>2</sup>. To turn the expression into ANF, it has to be bound like so: `let a = 0 in let b = n == a in ...`. The next problem lies within `Cons(n, build(n - 1))`. Here, the function call `build(n - 1)` is not bound before use, which can be fixed in the following way: `let e = build(n - 1) in let f = Cons(n, e) in ...`. However, the code is still not in ANF form. The expression that was bound to `e` (i.e `build(n - 1)`) also has to be converted. This means that whenever something is converted to ANF we must not forget to also ANF convert the expression that is being bound. If this is done in the example above the result becomes: `let c = 1 in let d = n - c in let e = build(d) in let f = Cons(n, e) in ...`, which is finally in ANF form. See Fig. 2.5 for the full translation of Code 11.

Let us now define our intermediate representation *Stir* (see Fig. 2.6), which requires that the given program is in ANF form. We have implemented a procedure `anf : TypedProgram → Stir`, which takes a type checked program and translates it into *Stir* using continuations.

---

<sup>2</sup>Sometimes, compilers omit binding trivial constants such as integers, but we have decided to bind everything for the sake of uniformity

```

build n =
  let a = 0;
  let b = n == a;
  match b
    False ->
      let c = 1;
      let d = n - c;
      let e = build(d);
      let f = constrCons [n, e];
      ret f
    True ->
      let g = Nil;
      ret g
    
```

Figure 2.5: ANF representation of Code 11

Expr	$e$	::= int $n$   constr $n \bar{v}$   binop $o v_1 v_2$   call $f \bar{v}$   project $n v$   unboxed $\bar{v}$	simply an integer constructor with tag $n$ and a list of vars binary operation on two vars call function $f$ on a list of vars get the $n$ th field of some constructor an unboxed tuple with a list of vars
Body	$b$	::= ret $v$   let $v = e ; b$   match $v \bar{b}$	returns some var binds some var to $e$ in $b$ list of bodies we can branch to
Integer	$n$	::= int	an integer
Variable	$v$	::= identifier	a variable name
Operator	$o$	::= +   -   *   /   %   ==   !=   <   <=   >   =>	all operators used in GOOPEA
Function	$f$	::= identifier	top level function
Definition	$def$	::= decl $\bar{v} b$	a function declaration
Stir	$prog$	::= stir $\overline{decl}$	a program

Figure 2.6: A sequentially transformed intermediate representation: *Stir*

Now that we have a reasonable intermediate representation for our language, we will move on to the subject of managing memory in GOOPEA.

## 2.5.2 Memory Management

Different languages provide different levels of control over how memory is handled. In a lower-level language, suppose that we have a linked list allocated on the heap with the numbers 1, 2 and 3 (see Fig. 2.7), and we want to remove the node containing 2 from this list. A natural way to achieve this would be to use pointer manipulation such that we make the first node point to the third node instead, as seen in Fig. 2.7.

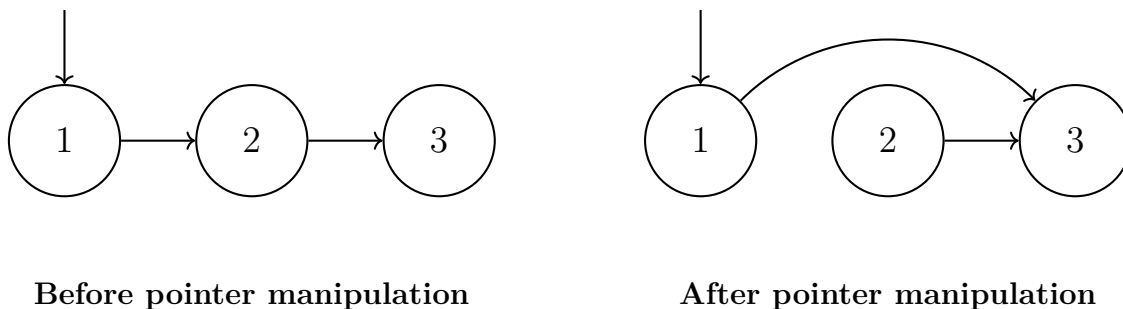


Figure 2.7: A visual representation of how removing the middle element of the linked list  $[1, 2, 3]$  could be done. For simplicity heap is imagined as a graph structure where nodes represent heap allocated regions of memory and an edge from one node to another represents that one such region holds a reference to another one.

At first this seems like a fine solution, but we have forgotten that the heap is persistent. This means that if something is allocated in the heap, the memory region will only be returned to the operating system when either the memory is deallocated or the process terminates.

Since only pointers have been updated during the removal, the memory corresponding to the 2 node has therefore not been returned to the operating system. This proves to be a problem because if nothing is done to the unreferenced memory it will continue to stay allocated until the program terminates. This is called a *memory leak*. Low-level languages like C puts all responsibility on the users in making sure they deallocate memory before it's no longer referenced. Other languages like Haskell or Java automate this process for the user with *garbage collection*.

## 2.5.3 Garbage Collection

Garbage collection is a form of automatic memory management that relieves the programmer of manual memory management [1, Ch. 7]. The most common type of garbage collectors are known as tracing collectors, which were introduced by McCarthy [19] in 1960 together with the “Mark and Sweep” algorithm. They are the primary memory management strategies in many modern high-level languages, including OCaml, Haskell and Java [17, 18, 9]. In such systems there exists a runtime system that automatically reclaims non-referenced objects to prevent memory leaks [1, Ch. 7].

Because GOOPEA is a high level language, we will similarly also need to use a garbage



```

1  dec(Heaped val) {
2      val.rc -= 1;
3      if val.rc == 0 {
4          for i in 0..val.size {
5              dec(val.data[i]);
6          }
7      }
8      dealloc(val);
9  }

1  inc(Heaped val) {
2      val.rc += 1;
3  }

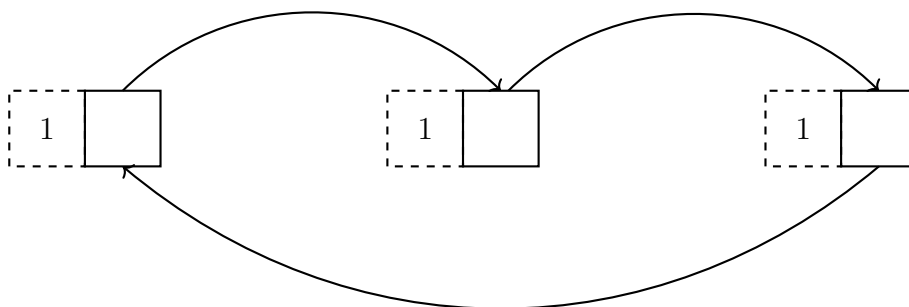
```

*Figure 2.9: Example implementations of `inc` and `dec` instructions given in pseudocode.*

code directly to C given reasonable implementations of `inc` and `dec`, which makes it an attractive approach if you are looking for a simple and elegant solution to memory management. Historically reference counting has never been known for its efficiency but new research has shown that it can be competitive with other tracing collectors [16].

To give the reader a more intuitive approach to how the reference counting instructions work, we provide the following pseudo-code to study for a deeper understanding:

In Fig. 2.8 the structures we saw had no cycles. Consider the cyclic memory structure shown in Fig. 2.10. Here we can see a fatal flaw when it comes to reference counting, which is that reference counting systems cannot naturally deal with cyclic memory references. What ends up happening is that the allocated regions keep each other alive so that none of them can get deallocated. This is a well known shortfall when using reference counting as garbage collection [1, Ch. 7].



*Figure 2.10: Cyclic data structure with no external references. The dashed-line boxes contains the reference counter, while the solid-lined boxes denotes some heap-allocated value*

Due to immutability guarantees and the absence of non-recursive bindings, such structures are not possible to create in GOOPEA, meaning reference counting is still feasible as a garbage collector.

### 2.5.5 Adding scope-based Reference Counting to our IR

Let us consider the extension to *Stir* given below, where we add two new sequencing instructions into the IR: `inc` and `dec`. They respectively represent an increment and likewise a decrement of the reference count of some heap allocation.

```

Body b ::= ...           defined above
         |  inc(v); b   increment var
         |  dec(v); b   decrement var

```

Adding these directly into structure of *Stir* turned out to be a good idea since the C representation and the previous IR before ANF were both too complex. This gives us a lot of power over where we can place our `inc/dec` instructions and how easily we can move them around if that is ever needed. The adaptability of this setup is something we will be using to our advantage later when we try to optimize the positions of these instructions. For now, we will use a simple scoped-based algorithm to place these instructions at appropriate points in the code.

We consider the transformation from *Stir* to *Stir<sub>RC</sub>* as a recursive function `insert : Body × Set<Var> → Body`. The main idea behind the procedure is that each function must clean up after itself. For decrements, this is quite clear. It simply means that whenever a function creates a heap-allocated value, the function should also be responsible for decrementing the reference count of value when the it goes out of scope (note that the reference count always starts at 1 when the value is created). As an example, let us look at the *Stir* function `foo`:

**Before adding decrements**

```

foo x =
  let y = constrpair [x, x];
  let z = bar(y);
  ret z

```

**After adding decrements**

```

foo x =
  let y = constrpair [x, x];
  let z = bar(y);
  dec(y);
  ret z

```

Here we clearly see that `y` is decremented before it leaves the scope (as it should be). One might think that `z` also should be decremented, but since it is returned and therefore inserted into the caller's scope, we omit any decrements. As for increments, we want to emit increments of some reference when that reference is used. This applies both to unboxed tuples, data constructors and return statements of external variables<sup>3</sup>. The exception to this rule is when references are passed as arguments to functions. In that case, we do not want to increment the reference count since that is the job of the original holder of the reference (i.e. the scope where the reference was created). If we extend the previous example we get:

---

<sup>3</sup>If we return a variable that we were given from another function. The function will then effectively have two references to the value and thus we need to increment it.

**Before adding increments**

```
foo x =  
  let y = constrpair [x, x];  
  let z = bar(y);  
  dec(y);  
  ret z
```

**After adding increments**

```
foo x =  
  inc(x);  
  inc(x);  
  let y = constrpair [x, x];  
  let z = bar(y);  
  dec(y);  
  ret z
```

Since `x` is used twice in constructor, we need to increment it twice, but again `y` is not incremented because the `bar` function never takes responsibility for decrementing it.

### 2.5.6 Core & GOOPEA's Runtime Representations

*Core* is the internal representation of C code. It is a basic representation with assembly-like instructions. After we have added our reference counting instructions, we translate the *Rc* IR to *Core* using a recursive procedure `score : BodyRc → Def`, where *Core* is a list of such definitions. For an example of how the C code looks that GOOPEA emits, see Fig 2.11.

```
Value length(Value xs) {  
  Value match_var1 = 1 == xs;  
  if (match_var1) {  
    Value v1 = 1;  
    return v1;  
  }  
  else {  
    Value ys = ((void**) xs)[4];  
    Value v2 = 3;  
    Value v3 = length(ys);  
    Value v4 = v2 + v3 - 1;  
    return v4;  
  }  
}
```

*Figure 2.11: Emitted C code for the GOOPEA function `length` defined in Code. 7*

When looking at the example, you may notice that everything has been given the `Value` type. This is the runtime representation of almost all values in GOOPEA (excluding unboxed tuples, which we will get to later). `Value` is a 64-bit sized type which holds integers, heap-allocated values (as pointers to memory regions on the heap), and atoms (an atom in GOOPEA is a constructor with zero arguments, for instance: `Nil` or `Empty`).

However, this representation introduces an ambiguity: how can we separate the pointers from our other values? In GOOPEA, we solve this issue by shifting all non-pointer values left by one step and adding one, effectively mapping every number  $n$  to  $2n + 1$ . This way we can look at the least significant bit to determine whether a value is a pointer or not. This works because `malloc` returns pointers which are suitably aligned for any kind of type [15, Ch. 3.2.3.1], which means the pointer will be byte aligned by 8 since the biggest type in GOOPEA is 8 bytes. This is important because when the decrement instruction is called on some pointer with reference count 1, then we will need to decrement all values that this pointer is holding, and if we can't differentiate between integers and pointers, this would cause segmentation faults when we try to decrement the non-existing counters of atoms and integers.

Our heap-allocated values are an array of values, where the first three such values are special (see Fig. 2.12). The first value holds the constructor tag of the value, the second holds the size of the constructor, the third holds the reference count of the value, and the rest of the values in the array are simply the values that the constructor holds. Using three 64-bit values for our headers is not very memory efficient, but for the sake of simplicity and ease of implementation we stuck with this approach.

tag	size	refcount	value <sub>1</sub>	value <sub>2</sub>
1	2	1	$2 * 7 + 1$	<code>ptr<sub>xs</sub></code>

*Figure 2.12: Memory layout of the GOOPEA expression `Cons(7, xs)`. The topmost row shows the names of the fields, and the lowermost the value of the field.*

Unboxed tuples are represented as a struct of multiple values, and since they are unboxed they will always be stack allocated. One such struct is emitted into the resulting C code for each length of tuple that appears in the program.

One interesting issue with having shifted integers in our language is that the usual arithmetic operations no longer work as expected. For instance, if we try to naively add two integers in their shifted form we get  $(2m + 1) + (2n + 1) = 2(n + m) + 2$ , which is incorrect since we want the result to have the value  $2(n + m) + 1$  (as this is precisely  $n + m$  shifted). It is clear we need to add extra instructions for each of the different arithmetic operations so that they work in our shifted semantics, which adds some overhead when working with arithmetic operations.



# 3

## An Interesting Idea: FIP

Since every heap-allocated value carries a counter, we can leverage this information to optimize updates. When the counter equals one, the value is guaranteed to be unshared, meaning nothing else refers to it. In that situation, if we need to modify the value, we know that the only party affected by the change is the modifier itself. Consequently, rather than allocating fresh memory (which can be expensive) and copying over each field, we can safely perform an in-place, destructive update.

To make this optimization possible we will have to perform the following three transitions to our IR:

1. Add reuse (Section 3.1)
2. Add ownership (Section 3.2)
3. Improve reference counting (Section 3.3)

These transitions gives us the new pipeline of the code generation seen in Fig 2.4.

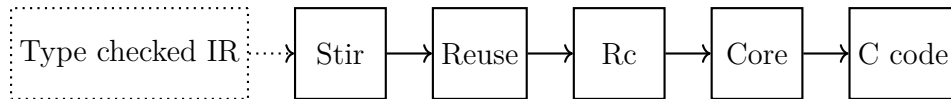


Figure 3.1: An extension of Fig. 2.4, The new pipeline of code generation

### 3.1 Extending the IR with Reuse

The goal of this transformation is to avoid deallocating and allocating values with the same arity<sup>1</sup> based on the resurrection hypothesis described by Ullrich and de Moura [24]. To achieve this goal we extend our IR with:

<b>Expr</b> <sub>Reuse</sub>	$e ::=$	...	defined above
		<b>reset</b> $v$	reset var
		<b>reuse</b> $v$ in <b>constr</b> $n \bar{v}$	reuse var in a constructor

The two new expressions **reset** and **reuse** come in pairs, which means that either both or neither are inserted at once. The **reset**  $v$  expression marks the variable  $v$  as reusable in a future **reuse** expression. The **reuse** instruction will then be able to reuse the memory of the previously **reset** variable if and only if the reference

<sup>1</sup>Arity is the number arguments taken by a function

count of the memory region is one. The algorithm for inserting these expressions is taken from the one introduced by Ullrich and de Moura [24], and consists of three functions:  $R$ ,  $D$  and  $S$  (see Fig. 3.2).

$$\begin{aligned}
 R &: \text{Body}_{Stir} \rightarrow \text{Body}_{Reuse} \\
 R(\text{let } x = e; b) &= \text{let } x = e; R(b) \\
 R(\text{ret } x) &= \text{ret } x \\
 R(\text{match } x \bar{b}) &= \text{match } x \overline{D(x, n_i, R(b_i))} \\
 &\quad \text{where } n_i = \text{arity of } x \text{ in the } i\text{th branch}
 \end{aligned}$$

$$\begin{aligned}
 D &: \text{Var} \times \mathbb{N} \times \text{Body}_{Reuse} \rightarrow \text{Body}_{Reuse} \\
 D(z, n, \text{match } x \bar{b}) &= \text{match } x \overline{D(z, n, b)} \\
 D(z, n, \text{ret } x) &= \text{ret } x \\
 D(z, n, \text{let } x = e; b) &= \begin{cases} \text{let } x = e; D(z, n, b) & \text{if } z \in e \text{ or } z \in b \\ \text{let } w = \text{reset } z; S(w, n, b) & \text{if } S(w, n, b) \neq b \text{ for fresh } w \\ \text{let } x = e; b & \text{otherwise} \end{cases}
 \end{aligned}$$

$$\begin{aligned}
 S &: \text{Var} \times \mathbb{N} \times \text{Body}_{Reuse} \rightarrow \text{Body}_{Reuse} \\
 S(w, n, \text{let } x = e; b) &= \begin{cases} \text{let } x = \text{reuse } w \text{ in constr}_i \bar{y}; b & \text{if } e = \text{constr}_i \bar{y}; b \wedge \text{length of } \bar{y} = n \\ \text{let } x = e; S(w, n, b) & \text{otherwise} \end{cases} \\
 S(w, n, \text{ret } x) &= \text{ret } x \\
 S(w, n, \text{match } x \bar{b}) &= \text{match } x \overline{S(w, n, b)}
 \end{aligned}$$

Figure 3.2: Functions for inserting reset/reuse pairs. Notice how reset is only added if reuse is successful

$D$  and  $S$  are auxiliary functions which aid  $R$  in performing the translation from  $\text{Body}_{Stir}$  to  $\text{Body}_{Reuse}$ .  $R$  aims to replace `constr` expressions with `reset/reuse` pairs. Whenever  $R$  encounters a match operation, it tries to insert `reset` and `reuse` for the variable  $v$  that was matched. For each arm of the match,  $R$  calls  $D$  and sends  $v$  and the arity  $n$  of the matched constructor as arguments.  $D$  then carries this information and finds the first location where  $v$  is dead (not used in the remaining function body) and calls  $S$ , passing along  $n$ .  $S$  attempts to find a matching constructor with arity  $n$  and if it succeeds then it replaces this constructor `constr`  $n$   $\bar{v}$  with `reuse`  $v$  in `constr`  $n$   $\bar{v}$ . If it fails in finding a matching constructor,  $D$  never modifies the function body.

An example of applying  $R$  to a function can be seen in Fig. 3.3.

```
reverseHelper list acc =
  match list
  Nil ->
    ret acc
  Cons ->
    let x = project 0 list;
    let xs = project 1 list;
    let v14 = constrCons [x, acc];
    let v15 = reverseHelper(xs, v14);
    ret v15

reverseHelper list acc =
  match list
  Nil ->
    ret acc
  Cons ->
    let x = project 0 list;
    let xs = project 1 list;
    let v19 = reset list;
    let v14 = reuse v19 in constrCons [x, acc];
    let v15 = reverseHelper(xs, v14);
    ret v15
```

*Figure 3.3: Applying R to (a pretty printed representation of) the `reverseHelper` function from Code 1, Above is before applying R, below is after*

When we apply R to `reverseHelper`, D starts looking for opportunities to reset and reuse `list` in both match arms. Since `list` is never used again after `let xs = project 1 list;`, S is called on the rest of the function body to look for constructor calls with two parameters. Naturally, it succeeds in finding such a constructor so D will insert the appropriate instructions and we end up with the transformation in the bottom of Fig. 3.3.

When we described our initial approach to reference counting we made no distinction between borrowed and owned values. This is because the idea of ownership doesn't inherently make sense in an immutable language. Since every reference is inherently read-only, you can safely have multiple references to the value without having to worry about any mutations occurring to the data. This allows all references to effectively behave like borrows, making an explicit ownership model unnecessary. However, ownership distinctions become very important when in-place updates are introduced. Suppose we pass a value into a function under the old semantics: the call does not increment the reference count because the argument is treated as borrowed. If the callee then reuses that value in-place, the original caller will see its data unexpectedly change, violating immutability guarantees. To prevent this, we must distinguish owned references from borrowed references (which remain immutable and always trigger a copy before mutation) which is what we will do in the following section.

## 3.2 Borrowing

In order to safely introduce in-place updates while preserving the language’s immutability guarantees, we need a mechanism to track which references exclusively own their data versus those that merely borrow it. We will need to go through each function and mark all parameters as either borrowed or owned so that we can treat them separately when we insert reference counting instructions. In order for us to be able to mark parameters as owned we will, for each defined function, use a recursive procedure `collect : Bodyreuse → Set<Var>` to collect variables that potentially need to be marked as owned. We then go through all of the function parameters and check if any of them is in the set, and if they are they will need to be marked as owned. If no function parameters had their ownership status revised then we are done, but if any of the parameters of any function had its ownership status modified, then we have to redo the entire process again as the `collect` procedure could give a different result now that at least one more parameter is owned. This process is repeated until no more changes in ownership status are made.

$$\begin{aligned}
 \text{collect} &: \text{Body}_{reuse} \rightarrow \text{Set}\langle\text{Var}\rangle \\
 \text{collect}(\text{let } \_ = \text{reset } v; b) &= \text{collect}(b) \cup \{v\} \\
 \text{collect}(\text{let } \_ = \text{call } f \bar{v}; b) &= \text{collect}(b) \cup \\
 &\quad \{v_i \in \bar{v} \mid \text{if parameter } i \text{ is owned in } f\} \\
 \text{collect}(\text{let } x = \text{project } i \ v; b) \\
 &= \begin{cases} \text{collect}(b) \cup \{v\} & \text{if } x \in \text{collect}(b) \\ \text{collect}(b) & \text{if } x \notin \text{collect}(b) \end{cases} \\
 \text{collect}(\text{let } \_ = \_ ; b) &= \text{collect}(b) \\
 \text{collect}(\text{ret } \_) &= \emptyset \\
 \text{collect}(\text{match } \_ \bar{b}) &= \bigcup_{b \in \bar{b}} \text{collect}(b)
 \end{aligned}$$

Figure 3.4: Collects all variable that must be marked as owned in a set

The `collect` procedure shown in Fig. 3.4 is quite similar to the one given by Ullrich and de Moura [24]. In general, we need to add variables to the collected set if they are ever reset (reused) or depend on reset values. The one caveat being function calls, where we need to go through the argument list and collect all variables whose corresponding parameter in the function is marked as owned.

Let us illustrate how our procedure works with an example. Consider the following two GOOPEA functions `foo` and `bar`:

```

1  fix Maybe : Maybe
2  foo(x, z) = match x {
3      Nothing: Nothing,
4      Just(y): Just(y * z)
5  };

1  MaybeList : Maybe
2  bar(xs) = match xs {
3      Nil: Nothing,
4      Cons(x, xx): foo x
5  };

```

Figure 3.5: We assume that some appropriate types `Maybe` and `MaybeList` are already defined in the above example.

If we use our compiler to compile the two functions to the *Reuse* stage, we get the two pretty printed examples shown in Fig. 3.6.

```

foo x z =
  match x
  Nothing ->
    let v1 = Nothing;
    ret v1
  Just ->
    let y = project 0 x;
    let v8 = reset x;
    let v2 = binop * y z;
    let v3 = reuse v8 in constrJust [v2];
    ret v3

bar xs =
  match xs
  Nil ->
    let v4 = Nothing;
    ret v4
  Cons ->
    let x = project 0 xs;
    let v5 = foo(x);
    ret v5

```

Figure 3.6: Notice how the compiler infers that  $x$  will be reused in the *foo* function.

Table 3.1: Step-by-step ownership inference.  $\mathcal{O}$  = Owned,  $\mathcal{B}$  = Borrowed.

Iter	Ownership (before)	collect(foo)	collect(bar)	Ownership (after)
0	foo $\rightarrow [x : \mathcal{B}, z : \mathcal{B}]$ bar $\rightarrow [xs : \mathcal{B}]$	$\{x\}$	$\{\}$	foo $\rightarrow [x : \mathcal{O}, z : \mathcal{B}]$ bar $\rightarrow [xs : \mathcal{B}]$
1	foo $\rightarrow [x : \mathcal{O}, z : \mathcal{B}]$ bar $\rightarrow [xs : \mathcal{B}]$	$\{x\}$	$\{xs\}$	foo $\rightarrow [x : \mathcal{O}, z : \mathcal{B}]$ bar $\rightarrow [xs : \mathcal{O}]$
2	foo $\rightarrow [x : \mathcal{O}, z : \mathcal{B}]$ bar $\rightarrow [xs : \mathcal{O}]$	$\{x\}$	$\{xs\}$	Same as before

After initializing all parameters as borrowed, we iteratively apply `collect`. In the first round,  $x$  must be owned in `foo` (since it is reset), triggering an ownership update. In the next round, this causes  $xs$  in `bar` to also be marked as owned. The third iteration yields no further changes, indicating that a fixpoint has been reached and that the process is done.

Now that we have a procedure for deducing the ownership-status of our function arguments, we can move on to the final part of adding memory reuse in GOOPEA, which is adding reference counting.

### 3.3 More Precise Reference Counting

When we add explicit memory reuse to our IR, the old scope-based strategy no longer works. If we want the ability to reuse memory, we also need to be able to give complete ownership to other functions. This means that we can no longer decrement every variable at the end of their scope<sup>2</sup>. Thus, we must completely rethink reference counting. Instead of thinking in terms of scope, we will start thinking in terms of liveness of our variables.

Let us explain the above with an example: consider `let xs = build(10) in sum(reverse(xs)) + 25`. Here, when `xs` is used in `reverse`, it is clear to the observer that there is only one reference to it and thus the value should be reused. However, for the compiler, it is not so simple. The compiler does not know if the right-hand side of the addition references `xs`, so before it gives sole ownership to `reverse`, the compiler must perform liveness analysis<sup>3</sup> on `xs` to verify that this truly is the last reference.

We define:  $\text{insert} : \text{Body}_{\text{Reuse}} \rightarrow \text{Body}_{\text{RC}}$ , which is based on the one introduced by Ullrich and de Moura [24]. This transformation is a bit involved, so we break it down with helpers. We also keep two global maps:

1.  $\alpha$  which associates each function with a list of ownership statuses where the  $i$ -th element in the list is the status of the  $i$ -th parameter of the function.
2.  $\beta$  which associates each variable with an ownership status. Initially,  $\beta$  will be assigned with the ownership status computed by the procedure from Table 3.1. If a variable is not found inside of the map then this means that the variable should be considered owned.

In total, we use four helper functions. We will only give an implementation for one of them, as the other three are quite trivial to implement.

- $\text{add\_inc} : \text{Var} \times \text{Set}\langle \text{Var} \rangle \times \text{Body} \rightarrow \text{Body}$   
This procedure takes a variable, a set of live variables and a body and will add an `inc` to the var in the body if the var is either borrowed or live. Thus if it is both owned and dead (not live), we will not increment the variable.
- $\text{add\_dec} : \text{Var} \times \text{Body} \rightarrow \text{Body}$   
This procedure takes a variable and a body and decrements the variable in the body if the variable isn't free<sup>4</sup> in the body and owned. Thus, we only decrement dead and owned variables.
- $\text{add\_decs} : [\text{Var}] \times \text{Body} \rightarrow \text{Body}$   
This is a small helper which folds over the `add_dec` procedure.
- $\text{apply} : [\text{Var}] \times [\mathcal{O} \mid \mathcal{B}] \times \text{Body} \rightarrow \text{Body}$   
This procedure is given together with the `insert` procedure in 3.7.

<sup>2</sup>If we do, then we must also extend their lifetimes. This is because they then must live until the end of their scopes, because otherwise they can not be decremented.

<sup>3</sup>In our IR, we perform liveness analysis on a variable by checking if it is free in the remainder of function.

<sup>4</sup>Free variables are variables that appear in a term (such as an expression, function body, or program) without being bound within that term.

Now that we are familiar with the helper functions we can describe the final algorithm `insert`.

$$\begin{aligned}
 &\text{insert} : \text{Body}_{\text{Reuse}} \rightarrow \text{Body}_{\text{Rc}} \\
 &\text{insert}(\text{ret } x) = \text{add\_inc}(\emptyset, \text{ret } x) \\
 &\text{insert}(\text{match } x \bar{b}) = \text{match } x \overline{\text{add\_decs}(\bar{y}, \text{insert}(b))} \\
 &\quad \text{where } \bar{y} = \text{FV}(\text{match } x \bar{b}) \\
 &\text{insert}(\text{let } y = \text{project } i \ x; \ b) \\
 &\quad = \begin{cases} \text{let } y = \text{project } i \ x; \ \text{inc } y; \ \text{add\_dec}(\text{insert}(b)) & \text{if } \beta(x) = \mathcal{O} \\ \text{let } y = \text{project } i \ x; \ \beta[y \mapsto \mathcal{B}]; \ \text{insert}(b) & \text{if } \beta(x) = \mathcal{B} \end{cases} \\
 &\text{insert}(\text{let } y = \text{reset } x; \ b) = \text{let } y = \text{reset } x; \ \text{insert}(b) \\
 &\text{insert}(\text{let } x = \text{call } f \ \bar{y}; \ b) = \text{apply}(\bar{y}, \alpha[f], \text{let } z = \text{call } f \ \bar{y}; \ \text{insert}(b)) \\
 &\text{insert}(\text{let } x = \text{constr}_i \ \bar{y}; \ b) = \text{apply}(\bar{y}, \bar{O}, \text{let } x = \text{constr}_i \ \bar{y}; \ \text{insert}(b)) \\
 &\text{insert}(\text{let } x = \text{reuse } z \ \text{in } \text{constr}_i \ \bar{y}; \ b) \\
 &\quad = \text{apply}(\bar{y}, \bar{O}, \text{let } x = \text{reuse } z \ \text{in } \text{constr}_i \ \bar{y}; \ \text{insert}(b)) \\
 \\
 &\text{apply} : [\text{Var}] \times [\mathcal{O} \mid \mathcal{B}] \times \text{Body} \rightarrow \text{Body} \\
 &\text{apply}(y \bar{y}, \mathcal{O} \bar{b}, \text{let } z = e; \ b) = \text{add\_inc}(\bar{y} \cup \text{FV}(b), \text{apply}(\bar{y}, \bar{b}, \text{let } z = e; \ b)) \\
 &\text{apply}(y \bar{y}, \mathcal{B} \bar{b}, \text{let } z = e; \ b) = \text{apply}(\bar{y}, \bar{b}, \text{let } z = e; \ \text{add\_dec}(b)) \\
 &\text{apply}([], \_, \text{let } z = e; \ b) = \text{let } z = e; \ b
 \end{aligned}$$

Figure 3.7: Inserting `inc` and `dec` instructions. Note that  $\mathcal{B}$  stands for borrowed and  $\beta$  maps variables to ownership status.  $\text{FV}(b)$  is the function that returns the free variables of a given function body  $b$ .

For an example of how `insert` works, view Fig. 3.8. It is an example of how the algorithm adds reference counting instructions to the `reverseHelper` example in Fig. 3.3.

```
reverseHelper list acc =
  match list
  Nil ->
    dec list;
    inc acc;
    ret acc
  Cons ->
    let x = Project 0 list;
    let xs = Project 1 list;
    inc xs;
    let v6 = reset list;
    inc acc;
    let v4 = reuse v6 in constrCons[x, acc];
    let v5 = reverseHelper(xs, v4);
    dec v4;
    ret v5
```

*Figure 3.8: insert applied to the reverseHelper function from Fig. 3.3*

# 4

## An Interpreter for GOOPEA

After the final IR has been generated by the compiler, the GOOPEA pipeline can proceed in two different directions: either compile the AST to C code, or pass the AST to the interpreter to run the code internally. This section describes the latter approach.

### 4.1 Overview

The interpreter first performs an almost direct mapping of the final AST to its own representation. This is only for the interpreter to be able to implement its own functions acting on the statements, and does not require further code generation. The purpose of the interpreter is to allow the playground to execute our GOOPEA code locally in a high-performance and controlled environment. One can also run and debug with the interpreter without the web playground. The interpreter is modeled in a way that makes experimenting with FIP easy, as it keeps precise tracking of runtime behavior and provides an interface to access internal data.

### 4.2 Runtime Architecture

To support direct execution of compiled GOOPEA programs, the interpreter contains a runtime architecture that models program state, memory, and control flow. Central to this is the Interpreter struct, which most importantly contains function definitions, a simulated heap, a statement queue for current execution, and scoped environments for local variables. Together, these components enable the interpreter to emulate a queue-based runtime system with memory reuse and reference counting.

All values in the interpreter are represented using a Data enum. They can either be a 64 bit signed integer or a pointer to a heap allocated memory block. This is a safe solution to differentiate between integers and pointers with rusts strong type system. Due to how the C code uses the least significant bit to differentiate between integers and pointers, the interpreter must cap its data to 63 bits to ensure consistency between the two. Simply ignoring the most significant bit only works with unsigned integers, so the interpreter cleverly copies the second most significant bit to the most significant. This makes the signed 64 bit value only able to represent integers in the signed 63 bit range.

The ADTs are modeled the same way as in the C code; each constructor consists of a heap allocated memory block containing a tag, a size, a reference count, and optionally any data contained by the constructor. All fields in the blocks use the Data enum, but the three headers are guaranteed to be integers.

## 4.3 Execution Model

The GOOPEA interpreter executes a program one statement at a time, following an imperative model built on a queue of intermediate statements. Each execution step evaluates a single statement, updating memory, variables, or the call stack as necessary. This section describes how the interpreter evaluates expressions, handles function calls, and performs memory operations.

### 4.3.1 Expression Evaluation

Evaluating expressions is a fundamental part in running a program. An expression always writes to either a variable, or to memory pointed to by a variable. To retrieve the underlying data in a variable, their identifier name is used as the key in the local variables hash map. One expression can contain up to three variables, that can all be integers, pointers, or indexes into data structures. The interpreter supports a few different expressions, each corresponding to a different type of assignment or computation, such as:

- `a := 1`, assigning a constant to a variable.
- `a := b`, copying a variable.
- `a := b[c]`, assigning from a field.
- `a[c] := b`, assigning to a field.
- `a := operator(b, c)`, assigning from a binary operation.
- `a := malloc(5)`, assigning a pointer value from a memory allocation.

### 4.3.2 Function Calls and Scoping

When a function call statement is executed, the program creates a new context of local variables for the function scope. When the function returns, the old context is restored. The interpreter handles a function call by pushing the current variable context to a stack, and creating a new with the function parameters along with the function body's defined variables. The current statements are also pushed to a stack, and are replaced with a copy of the function body's statements for the interpreter to execute. When a function returns, the return value is put into a register, and all variables are overwritten with the context being popped off the stack. The statements are also restored to that of the caller function. All function calls are followed by an expression assigning the return register to a variable, ensuring the result of a function is caught in a local variable.

### 4.3.3 Control Flow

In GOOPEA, the `match` keyword is used to handle branches in the program. In the final compiled AST, they are represented as a series of *if* and *else if* statements, where each branch recursively contains its body. The interpreter models this with one single statement, potentially containing multiple branches each with their own unique body. When such a statement is executed, the interpreter figures out which branch in the series should be taken in one step. It then pushes the body of the branch to the front of the statement queue, ensuring the branch is executed before returning back to the statements after the branch. This queue based control flow was chosen for its simplicity when the ability to save total state between each executable statement is needed.

## 4.4 Memory Management and FIP

Memory in the GOOPEA interpreter is managed by reference counting which provides automatic and deterministic memory de-allocations. To emulate how allocations works in C, the interpreter implements its own memory allocation algorithm which takes a size as an argument. The function searches the heap for any free memory locations, and creates a new vector consisting of 3+ size fields, to fit the three header values. A valid pointer to the newly allocated memory is returned for the program to use. When a reference count reaches zero, the memory object is removed and its location in memory is free to be allocated. FIP works just like in the C code. When a memory reuse is possible, the interpreter cleverly reuses memory that would be de-allocated for memory that would be allocated.

## 4.5 Debugging and Web Integration

As previously described, the interpreter executes a program by stepping through one statement at a time. This provides excellent debugging capabilities, but can be tedious to use in larger programs. The interpreter thereby implements a few functions that can jump larger gaps in the program. These are:

- (1) Run until memory is modified.
- (2) Run until a pointer is modified.
- (3) Step over a function.
- (4) Run until current function returns.
- (5) Run until the whole program has returned.

(1) Keeps the program running until anything is written to the heap. This is useful if you are only interested in your data structures on the heap. (2) differs from (1) in that it only pauses when a pointer in modified on the heap, or when memory is allocated or de-allocated. This function ignores integers assignments on the heap, focusing only on larger structural changes in your data. This is especially useful for visual comparisons between FIP and non FIP. (3) steps over a function, giving the debugger a way to efficiently pass large sections of code. In a series of function calls

on some data, this can be used to skip the first few to efficiently navigate to your function of interest. If one wants to return the current function, (4) is used. This skips all recursive function calls and returns to the current functions caller. When only the result of the whole program is of interest, (5) is used to let the interpreter finish executing all code.

### 4.6 Performance and Benchmarking

A system for benchmarking the interpreter was developed. It measures total execution time, average statements per second, and peak memory usage. It is important to note that performance is proportional to the speed of the computer running it, but the interpreter consistently operates at over a million statements per second on most laptops. There is a clear difference in peak memory usage between FIP and not FIP. In edge cases, FIP uses constant memory while non FIP memory usage grows linearly with an input parameter. In those cases, speed is significantly improved when using FIP.

# 5

## Putting Everything Together in a Web Playground

For the playground, three main pages were created. One for the editor, one for code snippets examples, and a final for the documentation and other information. These are the main pages on the websites because we believe that these fulfill the main requirements for the playground: the abilities to explore, experiment with, and learn about the language and FIP functionality. These pages can be navigated between using the navigation bar that sits across the top.

JS, HTML and CSS are often combined together and are a standard for web development, and choosing them would therefore mean a plethora of resources to refer to when developing, as opposed to purely Rust and WebAssembly, which are relatively immature. However, since it was decided early on that the back-end was to be written in Rust, something to connect the Rust and JS functionalities was needed. The online playground is written using JavaScript (JS), HTML, CSS and connects to the Rust back-end with *wasm-bindgen* [5] and *wasm-pack* [25].

Wasm-bindgen is a library that generates code to allow higher-level interactions between Rust and JS that offers this solution, meaning that you can call a JS function from Rust code and vice versa. For this project, the aspect of this that was desirable was that which lets Rust functions be labeled for export to JS and then used on the JS side to call on the interpreter and compiler. This is aided by *wasm-pack*, which builds and packages said Rust functions into a wasm binary. The Rust package *Basic-HTTP-Server* was also used, as a light-weight tool to create and run an HTTP server. The server is hosted locally, and conveniently displays saved changes to code as soon as the page is reloaded, meaning that any updates to the source code can quickly be checked for viability.

The following sections discuss the three pages mentioned above.

### 5.1 The Editor Page

To have even basic functionality on the editor page, we needed a code editor that could be placed in-line on the page. CodeMirror [10], which returns a code editor for the web, was chosen for the fact that, while it still had built in features, was lightweight as many more of its features were packaged as addons [11]. While the code editor's role is important, it is not overly complicated. Therefore, a more



Figure 5.1: The editor page with the Zipper tree example and output tab open after the run button is clicked

complex in-line editor for a browser with its steeper learning curve such as Monaco, the web version of VS Code [20], was simply unnecessary for our purposes.

The addons used for the webpage are active-line, close-brackets, comment, mark-selection, match-brackets, show-hint, simple-mode, and some of the functions defined in the CodeMirror approximation of sublime’s keymap. Active-line, close-brackets, and match-brackets make the editor function closer to other code editors and are mostly just visual aspects, as is mark-selection that had to be added to show cursor-highlight even over the error highlighting. Comment and the keymap are for keyboard shortcuts such as toggle line comment, shift line up/down, and duplicate line down. Simple mode allows for regular expression matching to add syntax highlighting, which, while not necessary to functionality, makes the code much easier to read and interpret.

Above the editor are four buttons: clear, compile, debug, and run. The clear button simply clears the editor and other textareas of text. The other three are what call the rust functions mentioned above.

An important thing to keep in mind while designing user interfaces is to match your interface to the user’s expectations [23], meaning that there was relative importance placed on making this editor similar enough in layout to other online playgrounds. As such, the code editor takes up only the leftmost half of the page, and the rightmost holds information that could be useful or interesting for the user to see while coding.

The left-side buttons all cause the right side to display the Output tab if the compiler returns an error and their own respective tabs if it doesn’t. This is in order to increase the flow of the webpage, because, for example, why would the user click

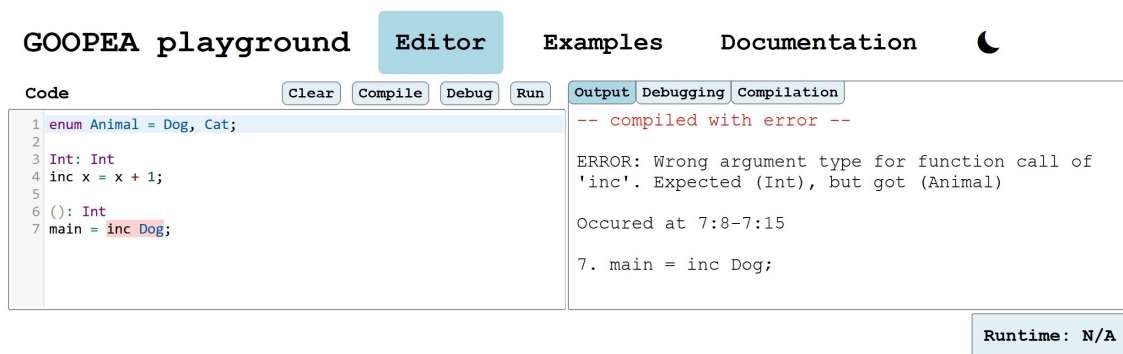


Figure 5.2: The output tab upon compilation error

*Debug* if they did not want to view the debugging tab. At the very bottom of the right side is a *Runtime* display. It only updates after the *Run* button is clicked, and displays the time that it takes to compile, start the interpreter, and run until done in milliseconds.

The right hand side has three main tabs: Output, Debugging, and Compilation.

### 5.1.1 Output

Output displays the output from the compiler, including printing and errors. All of the left-side buttons will print something in this field, though *Run* is the only one that will show anything beside compiler information. Note that the output's text view is the only one that is not a textarea. It is a *div* that is populated with several *span* in order to add color for clear visual cues: green for success, red for error.

Errors, returned from the compiler, include an error message that is displayed under the red message. If the error also includes a location, the editor's text is colored red using one of CodeMirror's built-in functions as seen in Fig. 5.2.

### 5.1.2 Debugging

Debugging has five extra buttons on the right side which controls the visualization: *step back*, *step forward*, *run until next return*, *run until next memory modification*, and *run until done*. All of these buttons call their respective Rust functions, those discussed in section 4.5, in order to manipulate the visualization discussed in section 5.1.4, but will not appear until the "debug" button on the left side is clicked.

### 5.1.3 Compilation

The Compilation tab shows all the different intermediate representations that the compiler creates between GOOPEA code and C code.

Outside of the compilation button, the page calls the compiler with the editor field's code every time the page loads and when there's a keyup event while the user is active in the CodeMirror editor. This is an approximation of continuous compilation and was chosen over a time-based approach because it only compiles when the user

## 5. Putting Everything Together in a Web Playground

GOOPEA playground **Editor** Examples Documentation

Code       **Compilation**

```

1 enum Tree =
2   Bin(Tree, Tree),
3   Tip Int;
4
5 enum TZipper =
6   Top,
7   BinL(TZipper, Tree),
8   BinR(Tree, TZipper);
9
10 fip (Tree, TZipper): Tree
11 down(t, ctx) =
12   match t {
13     Bin(l, r):
14       down(l, BinL(ctx, r)),
15       Tip x: app(Tip(x + 1), ctx)
16   };
17
18 fip (Tree, TZipper): Tree
19 app(t, ctx) =
20   match ctx {
21     Top: t,
22     BinR(l, up):
23       app(Bin(l, t), up),
24     BinL(up, r):
25       down(r, BinR(t, up))
26   };
27
28 fip Tree: Tree
29 tmap t = down(t, Top);
30
31 fip (): Tree
32 main = tmap(Bin(Tip 1, Bin(Tip 2, Tip 3)));

```

Select view 1: C code Select view 2: stir

```

#include <stdio.h>
#include <stdlib.h>

typedef __int64_t Value;

Value app(Value t, Value ctx);
Value down(Value t, Value ctx);
Value main();
Value tmap(Value t);

Value inc(Value ref) {
  if (!(1 & ref)) {
    void** ptr = ref;
    ptr[2]++;
  }
  return ref;
}

Value dec(Value ref) {
  if (!(1 & ref)) {
    void** ptr = ref;
    if (ptr[2] == 1) {
      for (int i = 3; i < ptr[1] + 3; i++) {
        dec(ptr[i]);
      }
    }
  }
  free(ref);
  return ref;
}

app t ctx =
  match ctx
  0 ->
    ret t
  1 ->
    let up = Proj(0, ctx);
    let r = Proj(1, ctx);
    let v1 = Ctor(2, t, up);
    let v2 = down(r, v1);
    ret v2
  2 ->
    let l = Proj(0, ctx);
    let up = Proj(1, ctx);
    let v3 = Ctor(0, l, t);
    let v4 = app(v3, up);
    ret v4

down t ctx =
  match t
  0 ->
    let l = Proj(0, t);
    let r = Proj(1, t);
    let v5 = Ctor(1, ctx, r);
    let v6 = down(l, v5);
    ret v6
  1 ->
    let x = Proj(0, t);
    let v7 = 1;
    let v8 = x + v7;
    let v9 = Ctor(1, v8);
    let v10 = app(v9, ctx);
    ret v10

```

Runtime: N/A

Figure 5.3: The diff view under the Compilation tab

GOOPEA playground **Editor** Examples Documentation

Code       **Compilation**

```

1 enum Tree =
2   Bin(Tree, Tree),
3   Tip Int;
4
5 enum TZipper =
6   Top,
7   BinL(TZipper, Tree),
8   BinR(Tree, TZipper);
9
10 fip (Tree, TZipper): Tree
11 down(t, ctx) =
12   match t {
13     Bin(l, r):
14       down(l, BinL(ctx, r)),
15       Tip x: app(Tip(x + 1), ctx)
16   };
17
18 fip (Tree, TZipper): Tree
19 app(t, ctx) =
20   match ctx {
21     Top: t,
22     BinR(l, up):
23       app(Bin(l, t), up),
24     BinL(up, r):
25       down(r, BinR(t, up))
26   };
27
28 fip Tree: Tree
29 tmap t = down(t, Top);
30
31 fip (): Tree
32 main = tmap(Bin(Tip 1, Bin(Tip 2, Tip 3)));

```

Step: C code Copy

```

#include <stdio.h>
#include <stdlib.h>

typedef __int64_t Value;

Value app(Value t, Value ctx);
Value down(Value t, Value ctx);
Value main();
Value tmap(Value t);

Value inc(Value ref) {
  if (!(1 & ref)) {
    void** ptr = ref;
    ptr[2]++;
  }
  return ref;
}

Value dec(Value ref) {
  if (!(1 & ref)) {
    void** ptr = ref;
    if (ptr[2] == 1) {
      for (int i = 3; i < ptr[1] + 3; i++) {
        dec(ptr[i]);
      }
      free(ref);
    }
  }
  else {
    ptr[2]--;
  }
  return ref;
}

```

Runtime: N/A

Figure 5.4: The code view under the Compilation tab

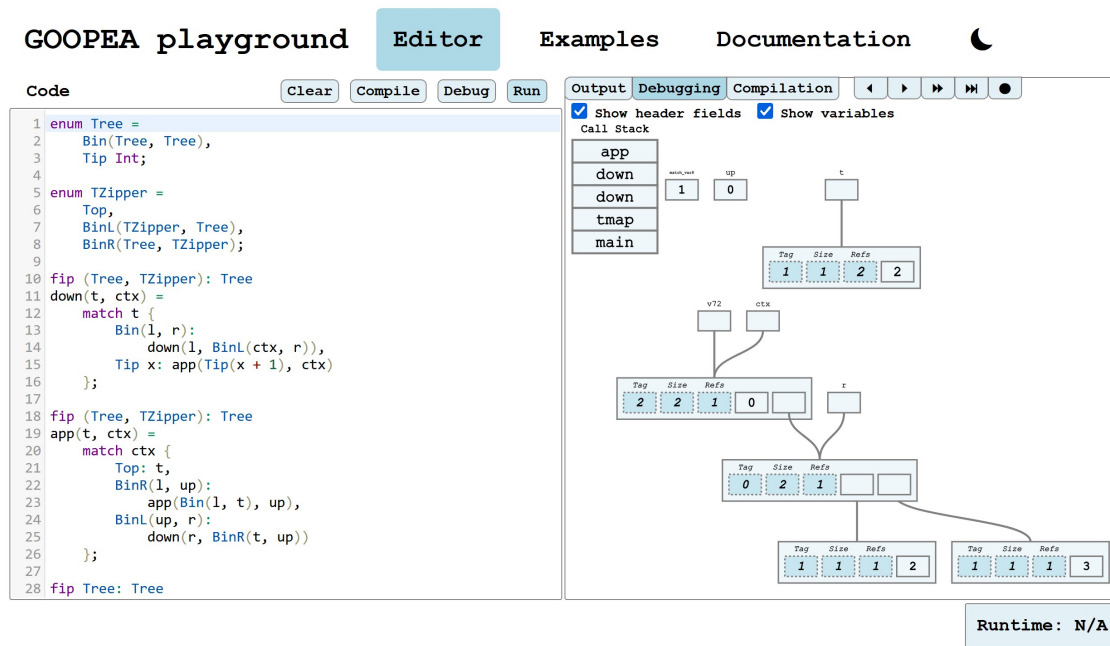


Figure 5.5: A step in the visualization of the Zipper tree example

is active, and time-based would simply continue without pause even if the page is open forever. This kind of continuous compilation is therefore unnecessary, as one would only want to recompile when the code is changed, and the easiest way of doing that without checking every time if the code has changed is to just do it when the user is actively typing.

### 5.1.4 Visualization

The key idea behind the visualization is to render the memory as a graph, as can be seen in Fig. 5.5. Each distinct memory allocation is represented by a box with a number of smaller boxes inside. The smaller boxes corresponds to the fields of the underlying data structure, and there are more for bigger allocations. Each field then contains some data. Variables are also represented as free floating “fields”, with a label above showing the name. What makes this a graph is that each data container, (i.e. the fields and variables) can either contain a number or a pointer. If it is a pointer the field then physically points to the referenced allocation. This allows structures like trees to be easily understood. There is also a call stack, giving some context to what function is currently being executed.

There are two main parts of the implementation of the visualization: the graphics (“front-end”) and the layouting (“back-end”). These were implemented with the help of the D3 library and the ELKjs library, respectively [3, 7]. D3 is a powerful library for doing all kinds of dynamic data visualizations. It has great support for making nice graphics, as well as doing transitions between different data sets. This makes it a great choice, since we wanted the visualization to be dynamic, easy to follow, visually pleasing, and capable of smoothly transitioning between each step of the program.

ELKjs is a JS wrapper for the powerful graph layouting library Eclipse Layout Kernel (ELK) [6]. It is a very mature library with lots of algorithms and possible customizations. Two particular important features that ELK supports are ports and an interactive mode. Ports allow you to specify where from a node the edge is coming from, which is used to make pointers originate from the specific field, and interactive mode makes the layouter take the previous location of nodes into account when laying out the new graph, which makes the visualization substantially more stable between steps, reducing big movements and restructurings.

The actual logic behind the visualization works by taking a memory snapshot from the interpreter, and then using that create the logical graph of the data. ELKjs is then fed this graph and returns a new graph where each node has been neatly positioned. The transition to this new graph, via D3, is then divided into three phases: exit, shift, enter. Exit animates out old nodes that are now removed, shift moves existing nodes and edges to their new position and zooms the camera, and enter lastly animates in the new nodes. These three phases make the transition easy to follow and nice looking. If there is nothing to add or remove to the visualization the corresponding transitions are skipped to increase snappiness.

There are also two check-boxes at the top of the visualization window. Toggled on, these cause the visualization to show the variables or the header fields of the allocations respectively. Each allocation actually has three extra header fields that contain some metadata about the allocation: tag (for differentiating different constructors of an ADT), size, and reference count. These fields are usually not very interesting for the user, and can create unnecessary visual clutter. Hiding variables can also be a benefit to visual clarity for some programs.

### 5.2 The Example Page

The example page is also split in half vertically. This decreases the visual change between the editor and example pages, which, along with HTML elements such as buttons and CSS margins and paddings that help keep the tops of the textareas aligned between pages, make it so that the only thing that looks like it changes between the pages is the text on top of the fields. This is because even when just counting that both pages have two text views, they are too similar to have obvious jumps in layout; most users would find it discordant. The left hand field is still a CodeMirror field, but it is read-only. There are several examples to choose from, and the user can select which one they would like to view from the drop-down menu above the field. Next to the field is a button that exports the selected example code to the editor page if the user would like to edit or inspect them further with the compiler. On the editor page after one such export, there is another button to the left of the clear button: *Restore code*. This is in case the export was accidental or regretted, and is only there until the next reload of the page. It can be seen in Fig. 5.1.

The right hand side contains a textarea that hold the output of the examples. Some examples have some comments on the output or the code. This can be seen in Fig. 5.6, which has a simple walkthrough of which methods are called with which

The screenshot shows the GOOPEA playground interface. At the top, there are navigation links: "GOOPEA playground", "Editor", "Examples" (highlighted), and "Documentation". Below these are two tabs: "View: Zipper tree" and "View in editor". The main content area is divided into two panes. The left pane, titled "Editor", contains Haskell code for a zipper tree. The right pane, titled "Output", shows the execution results, including a list of nodes and a detailed walkthrough of the operations performed on the tree.

```

1 enum Tree =
2   Bin(Tree, Tree),
3   Tip Int;
4
5 enum TZipper =
6   Top,
7   BinL(TZipper, Tree),
8   BinR(Tree, TZipper);
9
10 fip (Tree, TZipper): Tree
11 down(t, ctx) =
12   match t {
13     Bin(l, r):
14       down(l, BinL(ctx, r)),
15     Tip x: app(Tip(x + 1), ctx)
16   };
17
18 fip (Tree, TZipper): Tree
19 app(t, ctx) =
20   match ctx {
21     Top: t,
22     BinL(l, up):
23       app(Bin(l, t), up),
24     BinR(up, r):
25       down(r, BinR(t, up))
26   };
27
28 fip Tree: Tree
  
```

The output pane shows the following text:

```

[A: [A: 2], [A: [A: 3], [A: 4]]]
which is the same as Bin(Tip 2, Bin(Tip 3, Tip 4))

walkthrough (note: numbered Tips to keep track of them):
main = tmap(Bin(Tip1 1, Bin(Tip2 2, Tip3 3)));

down(Bin(Tip1 1, Bin(Tip2 2, Tip3 3)), Top)
  Bin l,r
    l = Tip1 1
    r = Bin(Tip2 2, Tip3 3)
  down(Tip1 1, BinL(Top, Bin(Tip2 2, Tip3 3)))
    Tip x
      x = 1
    app(Tip1 1+1, BinL(Top, Bin(Tip2 2, Tip3 3)))
      BinL up,r
        up = Top
        r = Bin(Tip2 2, Tip3 3)
    down(Bin(Tip2 2, Tip3 3), BinR(Tip1 2, Top))
      Bin l,r
        l = Tip2 2
        r = Tip3 3
    down(Tip2 2, BinL(BinR(Tip1 2, Top), Tip3 3))
      Tip x
        x = 2
    app(Tip2 2+1, BinL(BinR(Tip1 2, Top), Tip3 3))
      BinL up,r
        up = BinR(Tip1 2, Top)
  
```

Figure 5.6: The example page with the Zipper tree example open

values to get the final value for the Zipper tree example. As this side is essentially both so the user can immediately know the code's output and to keep the editor and example page layouts similar, its information is hard-coded.

### 5.3 The Documentation Page

The documentation page is at its core a table filled with buttons, text, and more tables. All lines of text with a plus or minus sign are buttons styled to look like headers through a common class that also dictates both the switching between signs and toggling the visibility of the content under those headers. The hidden content can contain more headers, tables, or plain text. The headers' ability to toggle visibility of their sub-content makes the list fully collapsible down to the main headers: *Introduction*, *Editor*, and *Language*.

The *Introduction* section contains definitions of, for example, functional programming and functional in-place. It is meant to give the user a little introduction to the concepts describing the language. The *Editor* section contains keyboard shortcuts and short descriptions of the examples. It could be expanded to include other information that pertains to the editor page to help the user streamline their performance. The *Language* section contains information about the syntax of GOOPEA, with headers such as function declaration and keywords. This section could also be expanded by way of including more in-depth descriptions.

### 5.4 Common code and shared classes

To keep the volume of JS and CSS code at a minimum with so many elements and possible actions, code that lumps together related actions is necessary. It is for this

reason that all buttons and button-like elements in the playground have a common class that decides, among other things, their color, text style, and border traits. This also serves to make the playground look more cohesive.

The theme button, mentioned again in section 5.4.1, toggles between default (light) and dark theme. It does this by way of a CSS attribute and variables. Upon clicking the theme button, the symbol changes to its opposite and it sets a document attribute to dark or default. In the shared CSS, the variables containing color codes are declared twice, once for root or default, and another for if the attribute is set to mean that dark theme is toggled. By then styling all elements on the webpage using the variables, the webpage efficiently changes theme upon button click. The method opposing this one, using color codes and adding a shared class to all color-changing elements, loads too slowly and often causes a longer “flash” of white upon loading a page. This flashing still happens with the current method, but it is mitigated both by the efficiency of variables and that larger page elements are hidden until after the page is loaded, as the flash is the visible effect of heavy elements that take a long time to load.

Any shared information between the pages or which state a page was in before being unloaded is stored in `localStorage`. The reason that `localStorage` was chosen over `sessionStorage` is that `sessionStorage` could sometimes lose information upon reloading the page via address bar, and certainly wouldn’t remember information after the tab was closed. Any elements that need to be updated from their original HTML state are accessed once the DOM (Document Object Model) has loaded.

All JS functions that call Rust code must be asynchronous as they *await* `wasm-bindgen`’s response to their query. Therefore, the methods that call on this method must be asynchronous as well. All three buttons that call Rust functions, mentioned in section 5.1, call asynchronous functions that await the shared function that sends the editor’s code to the compiler and populates all the values that will be shown in the textareas. Said function returns whether or not it received an error from the compiler, and therefore allows its callers to decide which tab and output to display.

### 5.4.1 The Navigation Bar

The Navigation bar is added to all the pages at the top of the HTML body. It contains the “logo”, three buttons representing the three pages of the website, and a theme toggle button. The logo is just the name of the website (“GOOPEA Playground”) on a button that takes the user back to the editor page. As the styling for the navigation bar is unanimous between the three pages, all shared styles beyond those for the navigation bar are also in the navigation bar’s CSS file. The most prominent shared style is for buttons, to ensure that they look the same between pages. Shared styles are incredibly important to a website both to save time by avoiding styling individual elements and to ensure that the website does not look disjointed.

The navigation buttons are declared separately on each page because each page has different elements it may need to save before unloading. For example, the editor

page needs to save its code and the example page needs to save the current example, which are two different processes, but the documentation page doesn't need to save anything and is simply loaded as it was coded in HTML. The function to change which page the user is viewing is, however, shared, and it is called by all state-saving functions that are triggered by the user clicking away from or reloading the page.

### 5.5 Summary of the playground's features

The editor page contains the primary editor in which the user can write their own code on the left and then compile, debug, or run it. The panel on the right hand side can display the output, the debugging visualization, or the various intermediate representations.

The example page contains an editor and associated drop-down menu from which the user can select an example to view, and if they want to edit it they can export it to the editor page via a button. On the right hand side the output and other helpful information is displayed.

The documentation page contains all the documentation on the language, some pertinent information about the primary editor found on the editor page, and an introduction to functional programming languages and the concept of "functional in-place". The documentation assumes that users have at least a low-level understanding of programming languages, but does explain basic concepts and syntax they will encounter in the language such as keywords and how functions are declared.



# 6

## Result, Discussion, and Conclusion

To start with, the performance of GOOPEA was evaluated by comparing the execution of programs compiled with scoped RC, improved RC, and FIP optimization. A range of benchmarks, including list traversal and binary tree operations, were tested on the interpreter. The metrics collected were execution time and peak memory usage. To slow memory allocations down so they are comparable with normal allocation instructions, all programs were tested with an artificial memory allocation delay of 0, 1, 2, 5, and 10 microseconds.

Across the benchmarks, programs where FIP optimizations were possible showed a significant decrease in memory usage, as well as a significant speed increase. Without artificial allocation delay, programs compiled with FIP performed the same or slightly worse than those without. See Appendix A for more detailed information.

Furthermore, benchmarks were done on a list-reversing program, as defined in Code 10, both with FIP and without FIP by directly compiling the C code with *GCC*[22] and evaluating the time it took with *hyperfine*[21]. The program was run on a list with 100 elements and was timed 10 times. FIP performed substantially better, as seen in Fig. 6.1, being about 67% faster. Two other benchmarks were also performed that gave similar results; FIP was about 67% better (see more details at Fig. A.6 and Fig. A.7.)

The results are in line with our theory that reducing the number of memory allocations decreases execution time. As observed, memory allocations are slow and the programs that require many memory modifications heavily favors FIP optimizations. With these benchmarks in consideration, it seems to us that FIP can be a very useful, effective, and desirable optimization for functional programming languages.

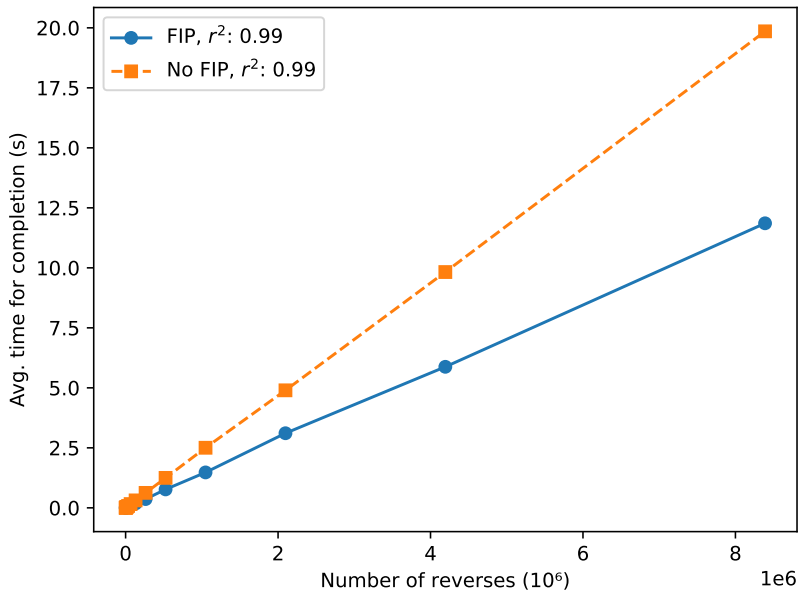


Figure 6.1: The graphical representation of the reverse example being compiled with and without FIP

Beyond performance, the implementation of the GOOPEA language and its integration with a web-based playground were successful in meeting the project’s goals. The language supports ADTs, pattern matching, and FIP optimizations. The web playground enables users to visualize memory usage of algorithms written in GOOPEA, with and without FIP. Together, the language and the playground form an accessible platform for experimenting with FIP in functional programming. The language is easy to use, and some quite sophisticated programs we have written are:

1. Building dynamic linked lists.
2. Common list manipulation functions such as concatenating and reversing.
3. Building balanced binary search trees.
4. Pseudo-random number generation.
5. List sorting.
6. Polynomial data structure with arithmetic and evaluation.

## 6.1 Future Work

In our implementation, we only allow memory reuse of constructors who has matching arity. In general, we can reuse any constructor whose arity is at least the original’s arity. This this allows for more memory reuse and potentially faster code at the cost of higher memory fragmentation.

In our experiments we observed that simple syntactic order can block memory reuse.

For example, in `sum(reverse(xs)) + length(xs)`, `reverse(xs)` cannot be reused since it is evaluated before `length(xs)`, whereas writing `length(xs) + sum(reverse(xs))` allows the temporary from `reverse(xs)` to be reused immediately. If the compiler knows that `(+)` is commutative then the compiler can detect it and optimize for it. Such an optimization could be implemented for many functions and could improve the general performance of reference counted programs and remove a the burden of remembering it from the programmer.

## 6.2 Societal and Ethical Aspects

Concerning the language, we do not foresee any direct major ethical or societal aspects violations. However, like all languages, the language will not limit a malicious actor from writing code which may cause harm. Furthermore, the language will most likely not be maintained after the end of the course. This could lead to potential vulnerabilities which were not accounted for, and thus remains the possibility for them to be exploited by malicious actors. We therefore advise users to use their own discretion to assess the risks when using the language.



# Bibliography

- [1] AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. *Compilers Principles, Technique, Tools*. Addison-Wesley, 2006.
- [2] APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, 1991.
- [3] BOSTOCK, M. D3, 2025. <https://d3js.org/>.
- [4] COLLINS, G. E. A method for overlapping and erasure of lists. *A method for overlapping and erasure of lists page 655-657* (1960).
- [5] CRICHTON, A. wasm-bindgen, 2025. <https://www.github.com/rustwasm/wasm-bindgen>.
- [6] Eclipse layout kernel (elk), 2025. <https://eclipse.dev/elk/>.
- [7] elkjs, 2025. <https://github.com/kieler/elkjs>.
- [8] FLAGANAN, C., SABRY, A., DUBA, B. F., AND FELLEISEN, M. The essence of compiling with continuations. *ACM SIGPLAN Notices, Volume 28, Issue 6* (1993).
- [9] GOSLING, J., JOY, B., STEELE, G., BRACHA, G., BUCKLEY, A., SMITH, D., AND BIERMAN, G. *The Java® Language Specification*. Oracle America, Inc, 2025, ch. Introduction.
- [10] HAVERBEKE, M., ET AL. Codemirror, 2025. <https://www.codemirror.net/5/>.
- [11] HAVERBEKE, M., ET AL. Codemirror manual, addons, 2025. <https://codemirror.net/5/doc/manual.html#addons>.
- [12] HIRSZ, M. Logos, 2025. <https://github.com/maciejhirsz/logos>.
- [13] HOPCROFT, MOTWANI, AND ULLMAN. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 2011, ch. Context-Free Grammars and Languages.
- [14] Lalrpop, 2025. <https://github.com/lalrpop/lalrpop>.
- [15] LOOSEMORE, S., STALLMAN, R. M., MCGRATH, R., ORAM, A., AND DREPPER, U. *The GNU C Library Reference Manual*, 2.41 ed. Free Software Foundation, 2024.
- [16] LOREN, A., LEIJEN, D., AND SWIERSTRA, W. Fully in-place functional programming. Tech. rep., Microsoft, 2023.
- [17] MADHAVAPEDDY, A., AND MINSKY, Y. *Real World OCaml: Functional pro-*

- gramming for the masses*. Cambridge University Press, 2022, ch. Understanding the Garbage Collector.
- [18] MARLOW, SIMON, ET AL. Haskell 2010 language report. *Available online* <https://www.haskell.org/onlinereport/haskell2010/>, retrieved 2025-05-15. (2010).
  - [19] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM* 3, 4 (1960), 184–195.
  - [20] MICROSOFT. Monaco editor, 2019. <https://www.github.com/microsoft/monaco-editor>.
  - [21] PETER, D. Hyperfine, 2025. <https://github.com/sharkdp/hyperfine>.
  - [22] STALLMAN, R. The gnu compiler collection, 2025. <https://gcc.gnu.org/>.
  - [23] TIDWELL, J., BREWER, C., AND VALENCIA, A. *Designing Interfaces : Patterns for Effective Interaction Design*. O’Reilly Media, Inc., 2020.
  - [24] ULLRICH, S., AND DE MOURA, L. Counting immutable beans. *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL’19)*. ACM, New York, NY, USA, 12 pages (2020).
  - [25] WILLIAMS, A. wasm-pack, 2024. <https://www.github.com/rustwasm/wasm-pack>.

# A

## Benchmarks

Interpreter benchmarks comparing scoped reference counting, improved reference counting (no FIP), and FIP. For a realistic environment where memory allocations are slower than basic instructions, different memory allocation delays were tested. All interpreter tests were run on a MacBook Air M1. Most benchmarks use a list library we wrote.

```
1  enum List = Nil, Cons(Int, List);
2
3  fip (List, List): List
4  reverseHelper(list, acc) = match list {
5      Nil: acc,
6      Cons(x, xs): reverseHelper(xs, Cons(x, acc))
7  };
8
9  fip List: List
10 reverseList list = reverseHelper(list, Nil);
11
12 fip (Int, Int): List
13 rangeList(start, stop) = match start <= stop {
14     True: Cons(start, rangeList((start + 1), stop)),
15     False: Nil
16 };
17
18 Int: Int
19 next x = x * 1664525 + 1013904223;
20
21 fip (Int, Int, Int): List
22 randList(seed, len, mod) = match len > 0 {
23     True: let a = next seed in
24         Cons((a % mod + mod) % mod, randList(a, len - 1, mod)),
25     False: Nil
26 };
```

*Code 12: Standard implementation of list functions used in benchmarks.*

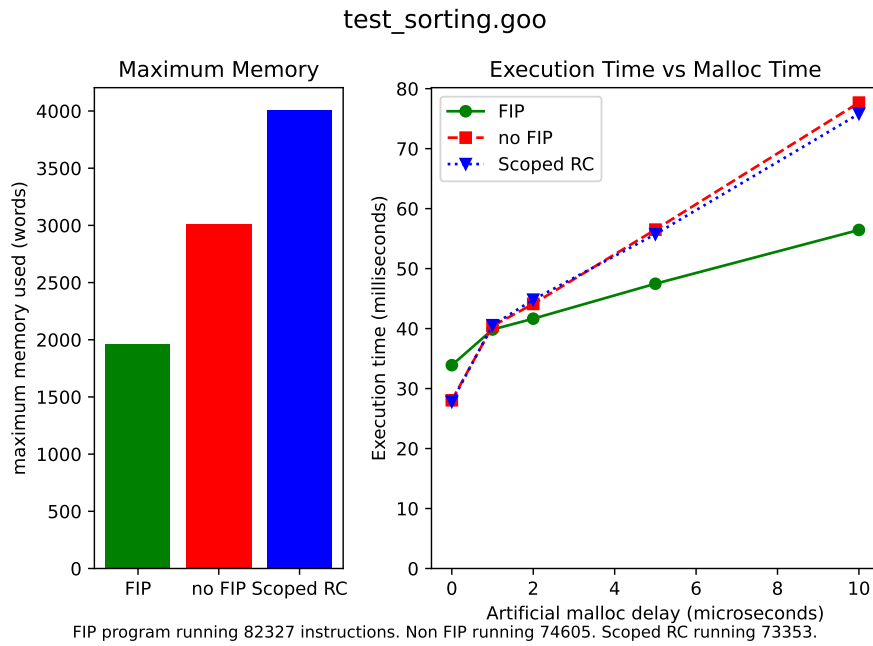


Figure A.1: Performance of merge sort on a list of 200 random integers.

```

1  fip (List, List): List
2  merge(a, b) = match a {
3    Cons(x, xs): match b {
4      Cons(y, ys): match x <= y {
5        True: Cons(x, merge(xs, b)),
6        False: Cons(y, merge(a, ys))
7      },
8      Nil: a
9    },
10   Nil: b
11 };
12 fip (List, List, List, Int): (List, List)
13 split(list, left, right, n) = match list {
14   Nil: (left, right),
15   Cons(x, xs): match n == 0 {
16     True: split(xs, Cons(x, left), right, 1),
17     False: split(xs, left, Cons(x, right), 0)
18   }
19 };
20 fip List: List
21 mergeSort(list) = match list {
22   Cons(_, xs): match xs {
23     Cons(_, _): let (left, right) = split(list, Nil, Nil, 0) in
24       merge(mergeSort left, mergeSort right),
25     Nil: list
26   },
27   Nil: Nil
28 };
29 (): List
30 main = mergeSort(randList(42, 200, 100));

```

## A. Benchmarks

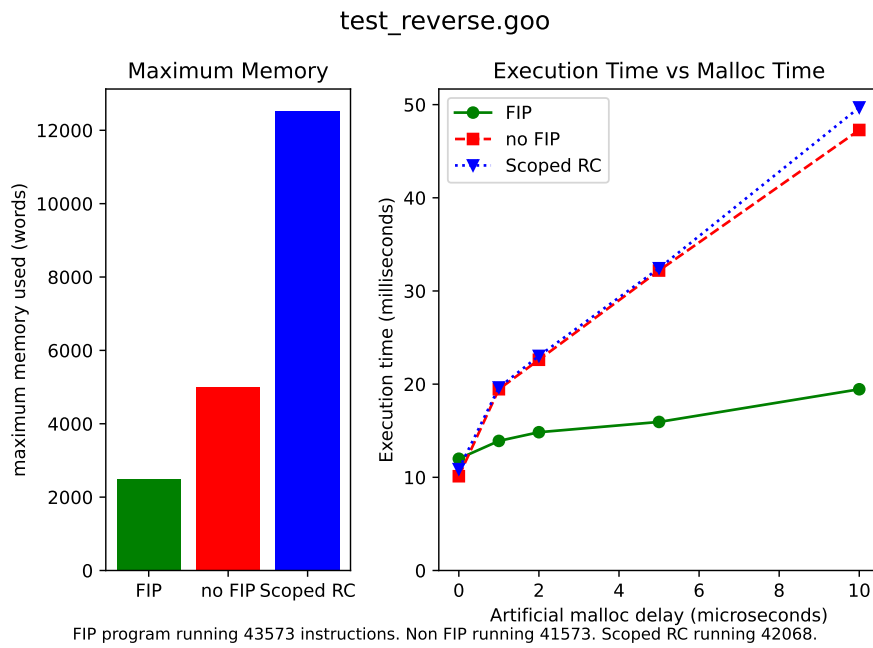


Figure A.2: Performance of a program reversing a list of 500 elements four times.

```
1 () : List
2 main = reverseList(reverseList(reverseList(reverseList(rangeList(1, 500)))));
```

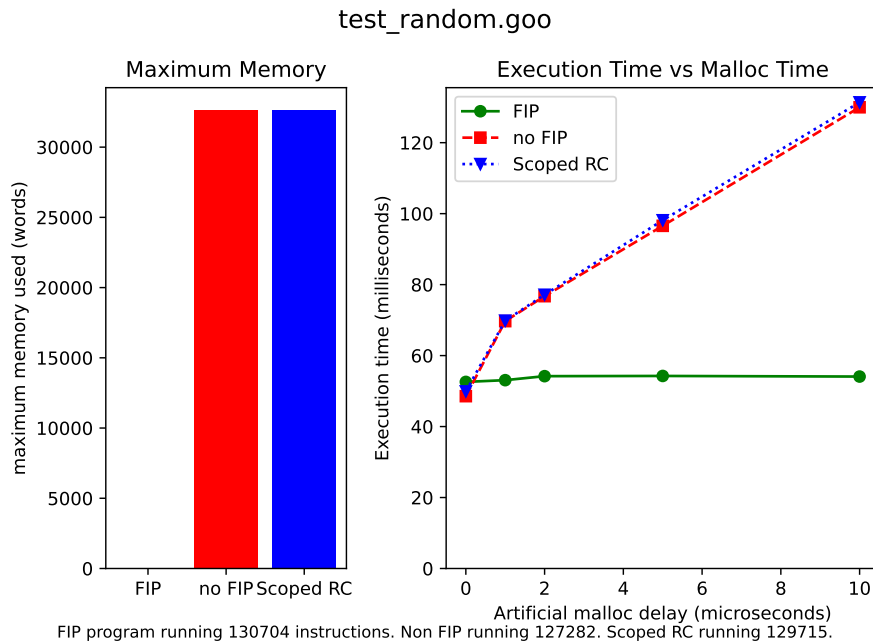


Figure A.3: Performance of a program counting the frequency of numbers modulo 10 in a list of 1000 random numbers. In this case, FIP optimization is able to remove all but 10 memory allocations.

```

1  enum Duallist = Nil, Cons(Int, Int, Duallist);
2
3  fip Int: Duallist
4  init(n) = match n > 0 {
5      False: Nil,
6      True: Cons(n - 1, 0, init(n - 1))
7  };
8
9  fip (Int, Duallist): Duallist
10 insert(x, list) = match list {
11     Nil: Nil,
12     Cons(n, y, xs): match x == n {
13         True: Cons(n, y + 1, xs),
14         False: Cons(n, y, insert(x, xs))
15     }
16 };
17
18 fip (Int, Int, Int, Duallist): Duallist
19 insertions(seed, n, mod, acc) = match n > 0 {
20     False: acc,
21     True: let x = next seed in let acc2 = insert((x % mod + mod) % mod, acc) in insertions(x, n
    - 1, mod, acc2)
22 };
23
24 fip (Int, Int, Int): Duallist
25 testRng(seed, n, mod) = let list = init(mod) in insertions(seed, n, mod, list);
26
27 (): Duallist
28 main = testRng(42, 1000, 10);

```

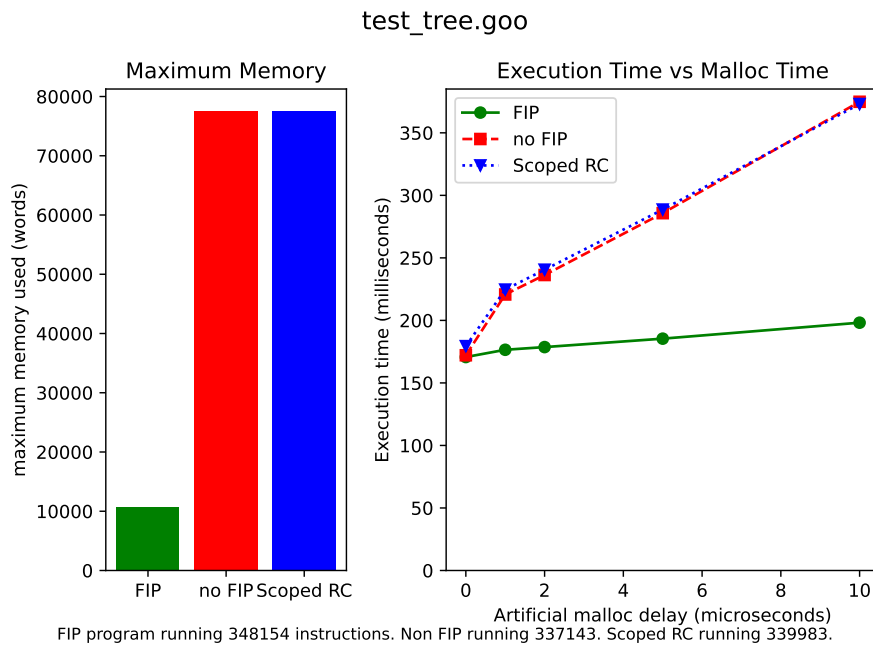


Figure A.4: Performance of a program inserting 1000 numbers to a binary tree without duplicates. Then it calculates the difference in sum between the original list and the tree.

```

1  enum BST = Empty, Node(BST, Int, BST);
2
3  fip (BST, Int): BST
4  insert(tree, value) = match tree {
5      Empty: Node(Empty, value, Empty),
6      Node(left, x, right): match value == x {
7          True: Node(left, x, right),
8          False: match value < x {
9              True: Node(insert(left, value), x, right),
10             False: Node(left, x, insert(right, value))
11         }
12     }
13 };
14
15 fip (BST, List): BST
16 insertList(tree, list) = match list {
17     Cons(x, xs): insertList(insert(tree, x), xs),
18     Nil: tree
19 };
20
21 fip BST: Int
22 sumTree(tree) = match tree {
23     Empty: 0,
24     Node(left, n, right): sumTree(left) + n + sumTree(right)
25 };
26
27 (): Int
28 main = let list = randList(42, 1000, 10000) in
29     let tree = insertList(Empty, list) in
30     sumList(list) - sumTree(tree);

```

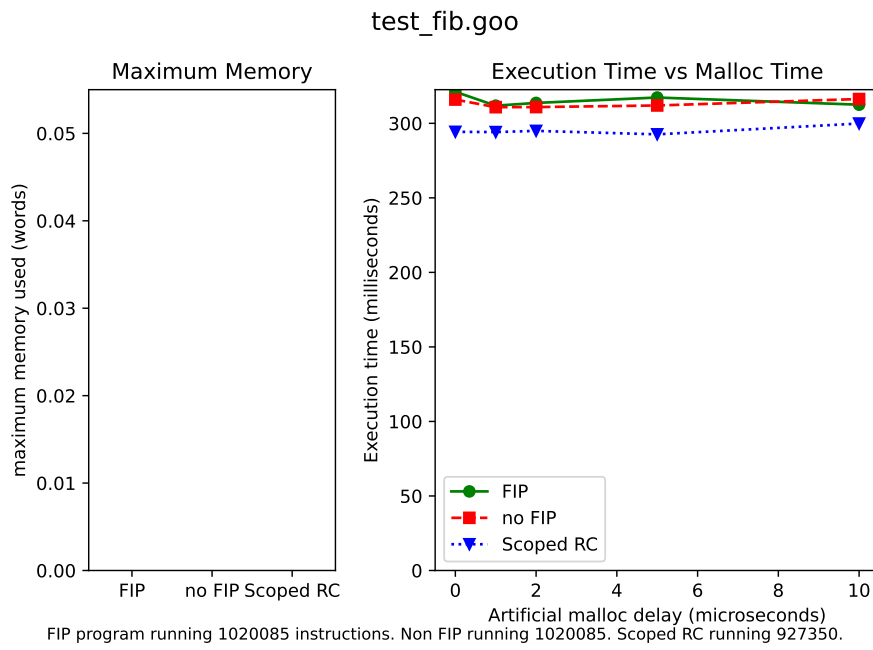


Figure A.5: Performance of a program calculating the 23:rd Fibonacci number recursively. This program uses no memory on the heap.

```

1  fib Int: Int
2  fib x = match x < 2 {
3      True: 1,
4      False: fib(x - 1) + fib(x - 2)
5  };
6
7  (): Int
8  main = fib(23);

```

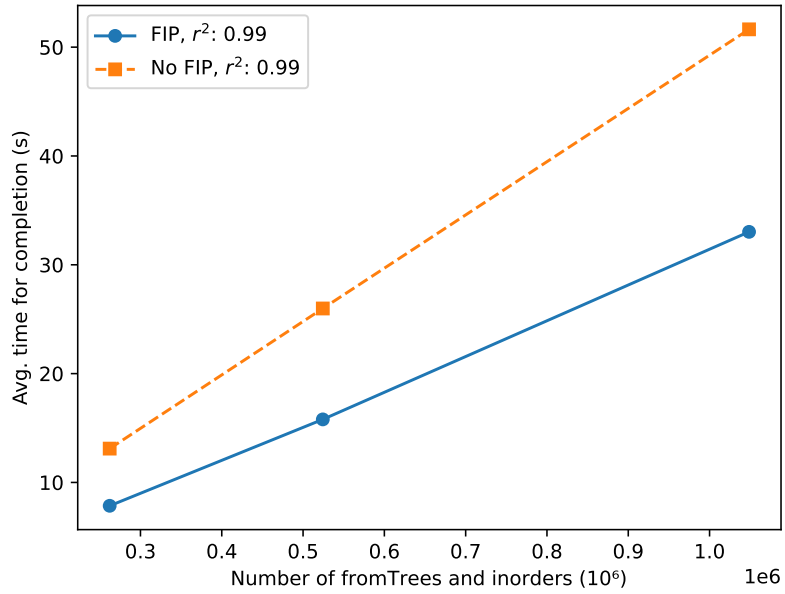


Figure A.6: Performance of creating a tree from a list and reverting it to a list  $n$  amount of times. The length of the list was 100.

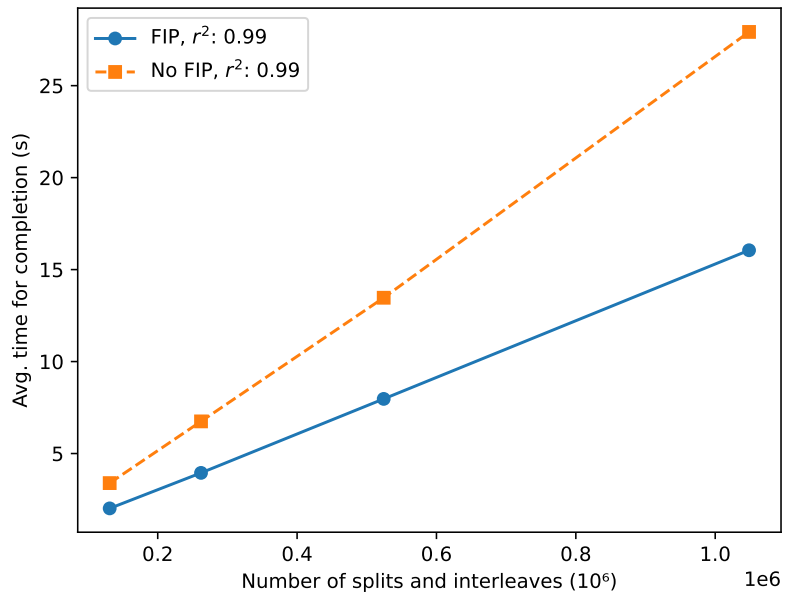


Figure A.7: Benchmarks of directly compiling GOOPEA to C and compiling the C codes with GCC.