Compiling a Functional Programming Language Using Combinators

Sumaia Aktar

Submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Department of Computer Science
Faculty of Mathematics and Science
Brock University
St. Catharines, Ontario

Abstract

Our work explores functional programming (FP), a paradigm grounded in lambda calculus and term rewriting that emphasizes immutability and pure functions, ensuring consistent outputs without side effects. We design and implement a strongly typed functional language that leverages FP principles to create robust, reliable code with inherent type safety. Our approach involves translating functional programs into a combinator language that bypasses variables and substitution. The combinators serve as higher-order constructs, transforming lambda terms into variable-free expressions to streamline execution. Subsequently, we develop a custom abstract machine specifically designed to execute combinator instructions, implemented in both Java and x86 assembler to generate an executable version that can run on actual hardware. By bridging high-level functional constructs with low-level executable code, this thesis demonstrates a clear pathway from functional programming concepts to practical machine-level computation including lazy and eager evaluation of language constructs. It highlights the efficiency and reliability of FP principles while laying the groundwork for future advancements in functional programming-based processors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Functional programming (FP) [3] languages, rooted in lambda calculus and beta-reduction (term rewriting), focus on using mathematical functions for computation. FP emphasizes immutability, where data cannot be changed once created, and pure functions that consistently produce the same output for the same input without side effects. In FP, functions are first-class citizens, meaning they can be passed as arguments, returned as values, which facilitate higher-order functions. Notable examples of FP languages include Haskell [2], Lisp [4], and Erlang [1]. In contrast, imperative programming languages are based on the von Neumann model [5], where instructions are executed sequentially to modify memory. These languages use mutable state, control structures (like loops and conditionals), and procedural programming to describe operations step-by-step. Object-oriented programming (OOP), a subset of imperative programming, organizes code around objects that encapsulate data and behavior, promoting modularity and code reuse. Examples of imperative and OOP languages include C, C++, Java, and Python. While FP languages emphasize what to solve through term rewriting and immutability, imperative languages focus on how to achieve a specific outcome through detailed state changes and procedural steps.

There are various methods to compile a functional language, with combinators being a notable approach. Combinators are advantageous because they avoid using variables and substitution (Section 2.1.4). If a lambda term lacks free variables (Section 2.1.4), its corresponding combinator term will also be variable-free. This results in simplified reduction (execution) by eliminating the need for substitution.

Our goals of the thesis:

a. Select an appropriate strongly typed functional language with primitive types and operations.

   - A strongly typed functional language is one in which every term has a type and operations are restricted to operands of compatible types. Such languages ensure type safety and help avoid type-related errors during compilation.These languages facilitate the creation of reliable and maintainable code by leveraging mathematical functions to perform computations. Our goal is to define such a language based on some primitive types such as integer with a suitable set of basic operations. For this reason our language will be a variant of Programming Computable Functions (PCF) by Gordon Plotkin in 1977 [15]

b. Translate into an appropriate combinator language.

   - Appropriate means that everything is correct and suitable within our context. A combinator language is one where programs (Section 2.1.3) are expressed using combinators (Section 3.1.1), which are higher order functions that can be combined to build more complex operations. More specifically, translation involves mapping the constructs of the functional language, i.e. programs, into combinators. This process helps to simplify functional terms into a form that can be more easily manipulated and executed. Due to the fact that the language has primitive operations such as addition, combinators corresponding to these operations need to be introduced.

c. Decide on an appropriate abstract machine that uses combinators as machine instructions so that this machine can be implemented in software or actually built as a custom processor for a computer based on the functional programming principle.

   - An abstract machine (Chapter 4) based on combinators would be designed to execute programs written in combinator form as defined in (Section 3.1.1). It would manage memory, execute instructions, and handle data according to the combinator-based representation of the program. This abstract machine could be implemented in software as a virtual machine, providing an environment to execute combinator-based programs. Alternatively, it could be physically built as a custom processor that directly executes combinator instructions, integrating with the functional programming principles.We have added primitive operations, and hence their corresponding combinators, to the programming language. These operations need their parameters computed before they can be executed (eager evaluation). On the other hand, the remaining combinators, such as combinator S and K, can be executed without computing the parameters first (lazy evaluation). Consequently, our abstract machine must be able to allow both ways of executing a combinator.

d. Implement steps a-c above in Java and, in addition, implement c in x86 assembler to obtain an executable version of the initial program.

- Implementing the strongly typed functional language in Java involves creating classes for types, functions, and combinators, and simulating the abstract machine's execution. For the x86 assembler implementation, the abstract machine would need to be translated into assembly language, where combinator instructions are encoded as machine code instructions. This would allow the program to be compiled into an executable binary, which can be run directly on x86 hardware or in an emulator.

1.1 Existing Work

Before delving into a detailed discussion of our own work, we next review relevant existing research related to our topic.

Jones [14] Describe the core methods for compiling functional languages, emphasizing the widespread use of graph reduction. Graph reduction is a technique where expressions are represented as directed graphs, allowing for efficient evaluation by reducing nodes step-by-step. This approach aligns well with the principles of functional programming, such as lazy evaluation and immutability, as it enables computations to be delayed and shared, reducing redundant evaluations and efficiently handling recursive calls and closures. In this thesis, we are interested in an approach using combinators. Combinators provide a way to represent functions without variables, which can simplify the compiler's task by reducing expressions based on function composition. By using combinator-based approaches, the compilation process avoids direct graph manipulation, potentially leading to optimizations that align well with functional programming's modularity and reusability. To sum up, Peyton Jones' exploration of graph reduction establishes a foundational technique for functional language compilers, while combinator-based approaches offer a promising direction for modular and efficient implementation.

Ben Lynn [10] proposed a Combinatory Compiler that translates expressions into basic combinators (S, K, and I), following a straightforward, minimalist approach. This implementation does not have primitive types and avoids the complexities of evaluation strategies, such as eager or lazy evaluation. In contrast, the work presented in this thesis introduces primitive types and supports lazy and eager evaluation, providing more flexibility and expressiveness in the language. While Lynn's approach simplifies compilation with a fixed set of combinators, this thesis expands the language to include richer-type systems and evaluation strategies, offering opportunities for optimization. Lazy evaluation reduces unnecessary computation, while eager evaluation can improve performance when immediate results are required, making the language more practical and adaptable to real-world scenarios.

Hudak and Kranz [7] define a language that is conceptually similar to the one explored in this thesis. However, the syntax and presentation appear somewhat outdated by modern standards. Notably, the language employs primitive types, with even structures such as lists treated as primitive types, in contrast to more recent functional languages where such types can be user-defined. Additionally, the execution of primitive operations in their approach is not derived generically; rather, it assumes a concrete implementation, which is directly used in the compilation process. Although the syntax and presentation of the language in their work is outdated by today's standards, it laid important groundwork for combinator-based compilation strategies. These earlier approaches were pivotal in demonstrating how functional programming languages could be compiled using a fixed set of combinators, serving as an inspiration for more modern techniques that allow for greater modularity and type flexibility. While the design of their work is simpler, focusing on concrete primitive implementations, the work in this thesis incorporates more dynamic type systems and flexible execution models, bridging the gap between early combinator-based languages and more complex modern functional languages.

Peter M. Maurer [12] investigates the use of extended combinators to translate functional languages into low-level data-flow graphs, emphasizing the elimination of variables through combinatory logic. Their work showcases how functional programming principles align with data-driven computation, focusing on exploiting parallelism in functional languages for data-flow architectures. In contrast, our thesis builds on similar concepts but shifts the focus to combinator-based abstract machines. By incorporating primitive operations and supporting both eager and lazy evaluations, our approach bridges the gap between theoretical combinatory logic and practical implementation, offering a more versatile execution model.

Hughes [8] introduces super-combinators, an extension of traditional combinators aimed at achieving full laziness and improving efficiency in graph reduction machines. By exporting maximal free expressions as parameters, super-combinators eliminate redundant copying during substitution, providing a more efficient alternative to Turner's combinators. While Hughes focuses on optimizing graph-reduction implementations, our work extends these ideas to combinator-based abstract machines, incorporating both eager and lazy evaluation strategies. Additionally, we integrate primitive operations, offering a more expressive and versatile execution model that bridges combinatory logic with practical applications in software and hardware.

Gibert [6] introduces a combinator-based framework for functional programming, focusing on efficient symbolic computation and parallelism through an algebraic model. His work highlights the design of the JAMachine, an architecture capable of processing combinatory logic in both sequential and parallel modes, demonstrating the potential of non-von Neumann computational paradigms. While Gibert's approach emphasizes algebraic semantics and symbolic computation, our thesis builds upon these foundations by incorporating primitive operations into the combinator framework and developing an abstract machine that supports both lazy and eager evaluation strategies. Furthermore, we extend the practical applicability of combinator-based computation by proposing implementations in both software and hardware, bridging the gap between theoretical models and real-world execution environments.

1.2 Methodology

Below is an overview of the methodology we employ in our work:

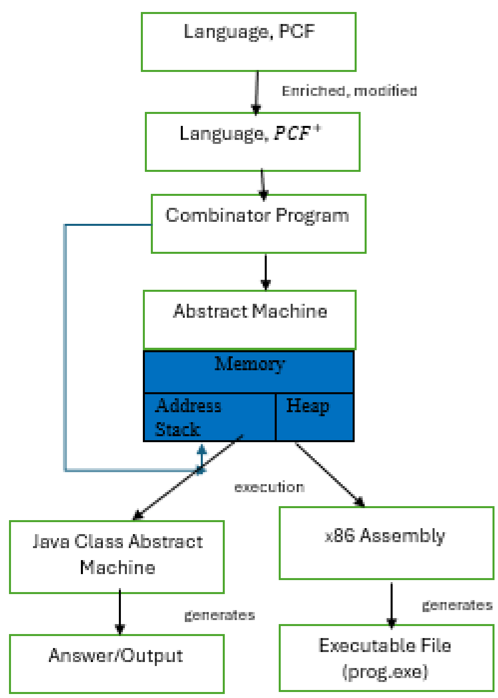

Figure 1.1: Typing of a PCFTerm.

Chapter 2

Programming Language

As our goal is to compile basic functional programming language using combinators, we have chosen an extended version of PCF (Programming Computable Functions), a functional programming language designed to express computable functions [15]. PCF is rooted in Scott's Logic of Computable Functions (LCF)[13]. PCF emphasizes mathematical rigor and theoretical foundations. Unlike LCF [13], which is an interactive automated theorem prover based on the theoretical foundation of the logic of computable functions, PCF focuses exclusively on terms, with a subset of these terms designated as PCF Terms. This emphasis on terms and computability makes PCF a powerful tool for exploring the theoretical aspects of programming languages and computational logic.Our extension PCF$^+$ adds the type Bool of Boolean values to PCF and replaces the Nat of natural numbers by the type Int of integers. Furthermore, we add several basic operations on Booleans and natural numbers such as equality test, addition, and multiplication as primitives. Adding these primitives allows a more efficient implementation and translation of these operations to corresponding combinators, and, later, to machine code.

2.1 Syntax

PCF$^+$, as its base language PCF, is a strongly typed programming language. In order to define its syntax, we first start to introduce the syntax of types.

2.1.1 Types

The syntax of types is defined by the following constructions:

- Int

- $Bool$
- If V is a set of type variables, then every element of V is a type.
- If σ and τ are types, then $\sigma \rightarrow \tau$ is a type.

For example, $Int \rightarrow Bool$ and $((Int \rightarrow Int) \rightarrow Bool)$ are types. Furthermore, if $a \in V$, i.e., a is a type variable, then $a \rightarrow Int$ is a type

2.1.2 PCFTerms

In order to define PCFTerms we assume that there is a constant symbol $\underline{n}$ for every integer n. Please note that a clear difference between the number n and the symbol $\underline{n}$ representing the number is the syntax of the language. However, if it is clear from the context we will often use the same notation 5 for the number and its textual representation. We have defined various operations to construct a PCFterm. These operations include:

- Variables: a variable i.e x is a term.
- NamedTerms: a NamedTerm t is a term, if t is a declaration (Section 2.1.3).
- Integer Constants: $\underline{\text{n}}$ for every integer n, i.e $\underline{2}$ or $\underline{5}$ is a term.
- Boolean Constants (True, False): $True$ and $False$ are terms.
- Integer Operations (Addition, Subtraction, Multiplication): If t_1 and t_2 are terms, then $t_1 + t_2$, $t_1 - t_2$, and $t_1 * t_2$ are also terms.
- Comparisons (Equal, LessEqual): f t_1 and t_2 are terms, then $t_1 = t_2$ and $t_1 <= t_2$ are also terms.
- Not operation: if t is a term, then $not\ t$ is a term.
- Boolean operations (and, Or): If t_1 and t_2 are terms, then $t_1\ and\ t_2$ and $t_1\ or\ t_2$ are two boolean terms.
- Conditional: (if...then...else) If t_1, t_2 and t_3 are some terms then $if\ t_1\ then\ t_2\ else\ t_3$ is a term.
- Abstraction: If x is a variable and t a term, then $\lambda x.t$ is a term.
- Recursion: If x is a variable and t a term, then $recx.t$ is a term.

- Application: If t_1 is a term and t_2 is another term, then $t_1 t_2$ is a term.

Variables, NamedTerms and constants, the smallest terms, are the fundamental building blocks of a PCFTerm. Some examples of PCFTerms are shown below:

- $\lambda x.\, t+1$
- $if\ x \leq (3+2\cdot x)\ then\ True\ and\ y\ else\ (not\ False\ and\ True)$
- $recf.\lambda n.if\ n=0\ then\ 1\ else\ n*f(n-1)$

The above examples illustrate abstraction, conditional, and recursive terms respectively. The first example demonstrates the smallest abstraction term with the variable x and the body $t+1$. The second example showcases a conditional term containing a subPCFTerm (another PCFTerm), $(3+2\cdot x)$, which demonstrates the addition of an integer constant and a multiplication term, where the multiplication is between an integer and a variable. The third example presents a recursive term involving the variable n and a body that includes another subPCFTerm with a conditional term.

Please note that we call a term a PCFTerm if it is typeable, a concept that we will define in the following sections.

2.1.3 Programs

A program consists of one or more declarations (Section 2.1.3). Each declaration, as will be discussed in the subsequent section, has three components: a name, a body, and a type. Similar to Java, a PCF$^+$ program must include at least one declaration named $main$, which marks the starting point for the execution. If the $main$ declaration is absent or incorrectly defined, the program will fail to execute.

$$\begin{array}{lll} \texttt{fact} & \texttt{::} & \texttt{Int} \rightarrow \texttt{Int} = \texttt{recf.}\,\lambda n.\,\texttt{if}\ n=0\ \texttt{then 1 else}\ n\times f(n-1); \\ \texttt{succ} & \texttt{::} & \texttt{Int} \rightarrow \texttt{Int} = \lambda n.\, n+1; \\ \texttt{main} & \texttt{::} & \texttt{Int = fact(succ 4);} \end{array} \tag{2.1}$$

Equation 2.1 illustrates a sample PCF$^+$ program containing three declarations: $fact$, $succ$, and $main$. Among these, the $main$ declaration signifies where the execution of the program begins.

Declaration

A declaration is the fundamental building block of a Program, consisting of three components: the $name$, the $type$, and the $body$. The body of the declaration is itself a $PCFTerm$ (Section 2.1.2). The basic structure of a declaration is as follows:

$$\text{Name :: Type} = \text{PCFTerm}$$

This means that a declaration begins with the $name$, which refers to the name of the declaration, followed by the $type$, and then the $PCFTerm$ that serves as the body of the declaration.

2.1.4 Typing Rules

We present the type system of PCF$^+$ by providing a set of typing rules specific for every term construction of the previous section. After this we will describe our implementation of type checking, which computes the most general type of each term and then compares this with the type given. A context Γ is a sequence consisting of pairs $x : A$ where 'x' represents a variable name and 'A' represents a type, ensuring that each variable name is unique within the list. We will write $\emptyset$ for the empty context, $a : A \in \Gamma$ if $a : A$ is in the sequence Γ, and $\Gamma, x : A$ for an arbitrary sequence that contains $x : A$. Typing judgments for terms within such a context are then defined using a standard approach for typing. Using these rules, i.e., deriving a type judgment $\Gamma \vdash t : A$, means to build a tree using the rules with the statement above at the root. Notice that the leaves of the tree must be rules with no predecessor (first four rules).

We call a term t typeable iff there is a context Γ and a type A so that the judgment $\Gamma \vdash t : A$ can be derived. Please note that if t is closed and typeable, then there is a derivation $\emptyset \vdash t : A$ for some type A. Now, add an example. Use $y : Int \vdash \lambda x.x + y : Int \to Int$. Use the proof package to build up the tree that ends with the statement above at the root.

$$\dfrac{\dfrac{x : Int \quad y : Int \quad x + y : Int}{x : Int, y : Int \vdash x + y : Int}}{y : Int \vdash \lambda x.\, x + y : Int \to Int}$$

The construction of the proof tree above ensures logical validity by systematically applying the above typing rules with integers and booleans. At the root of the tree, the lambda abstraction rule is employed to establish that $\Gamma \vdash \lambda x.x + y : Int \to Int$. This requires demonstrating that $y : Int, x : Int \vdash x + y : Int$. To achieve this, the integer operation rule for addition is used at the first level of the tree. This rule necessitates proving that both x

Table 2.1: Typing Rules

Name	**Rule**
Variables	$\Gamma \vdash x : A$ if $x : A \in \Gamma$
NamedTerms	$\Gamma \vdash x : A$ if $x : A \in \Gamma$
Integer constants	$\Gamma \vdash n : Int$ (for any integer constant n)
Boolean constants	$\Gamma \vdash True : Bool \quad \Gamma \vdash False : Bool$
Integer operations	$\dfrac{\Gamma \vdash t_1 : Int \quad \Gamma \vdash t_2 : Int}{\Gamma \vdash t_1 \oplus t_2 : Int}$ $\oplus \in \{+, *, -\}$
Comparisons	$\dfrac{\Gamma \vdash t_1 : Int \quad \Gamma \vdash t_2 : Int}{\Gamma \vdash t_1 \oplus t_2 : Bool}$ $\oplus \in \{\leq, =\}$
Not operation	$\dfrac{\Gamma \vdash t_1 : Bool}{\Gamma \vdash \neg t_1 : Bool}$
Boolean operations	$\dfrac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : Bool}{\Gamma \vdash t_1 \oplus t_2 : Bool}$ $\oplus \in \{\vee, \wedge\}$
Conditional	$\dfrac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : A \quad \Gamma \vdash t_3 : A}{\Gamma \vdash if \quad t_1 \quad then \quad t_2 \quad else \quad t_3 : A}$
Abstraction	$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$
Recursion	$\dfrac{\Gamma, x : A \vdash t : A}{\Gamma \vdash recx.t : A}$
Application	$\dfrac{\Gamma \vdash t_1 : A \to B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B}$

and y are of type Int. At the second level, the tree verifies these types individually through two sub-proofs: $x : Int \vdash x : Int$ and $y : Int \vdash y : Int$. Each sub-proof utilizes the variable rule to confirm that x and y are correctly typed as Int in their respective contexts. By following this structured approach, the proof tree adheres to the formal typing rules as listed above and logically demonstrates that $\lambda x.x + y$ is a function from Int to Int given that y is of type Int.

Please note that a term can have more than one type. For example, for the term $\lambda x.x$ and $\Gamma = \emptyset$, we can find derivations of $\Gamma \vdash \lambda x.x : Bool \to Bool$ and $\Gamma \vdash \lambda x.x : Int \to Int$.

However, For our typing algorithm and the semantics of PCF$^+$, it is essential to understand the concept of term substitution. Before delving into this topic, we need to grasp the notion of free variables.

Free Variable (FV)

The free variables of a term in PCF$^+$ are those variables that are not bound by any abstraction within that term. The set of free variables for any given expression can be defined by the following rules:

- For a variable x, the set of free variables is simply $\{x\}$.

- For a NamedTerm x, the set of free variables is empty.
- The integer and boolean constants have an empty set of free variables.
- For an addition $t_1 + t_2$, the free variables are the variables from t_1 union the free variables from t_2.
- For a subtraction $t_1 - t_2$, the free variables are the variables from t_1 union the free variables from t_2.
- For a multiplication $t_1 * t_2$, the free variables are the variables from t_1 union the free variables from t_2.
- For an eEqual $t_1 = t_2$, the free variables are the variables from t_1 union the free variables from t_2.
- For an lequal $t_1 \leq t_2$, the free variables are the variables from t_1 union the free variables from t_2.
- For a not $not\ t$, the free variables are the set of free variables of t.
- For an and $t_1 \wedge t_2$, the free variables are the variables from t_1 union the free variables from t_2.
- For an or $t_1 \vee t_2$, the free variables are the variables from t_1 union the free variables from t_2.
- For an abstraction $\lambda x.t$, the set of free variables is the set of free variables of t, excluding x.
- For an application $t_1\, t_2$, the set of free variables is the union of the free variables of t_1 and the free variables of t_2.

For instance, the lambda term $\lambda t.\, t$, which represents the identity function, has no free variables. However, the term $\lambda t.\, t_1$ contains the free variable t_1, since t_1 is not bound within the term.

Substitution

Substitution involves replacing a variable x in a term t with another term t', denoted as $t[t'/x]$. In the context of PCF$^+$, substitution is defined recursively based on the structure of terms as follows (note: x and y are variables, while t_1 and t_2 are any PCF$^+$):

$$
\begin{aligned}
&x[x := t'] = t[x := t'] \\
&(t_1 + t_2)[x := t'] = t_1[x := t'] + t_2[x := t'] \\
&(t_1 - t_2)[x := t'] = t_1[x := t'] - t_2[x := t'] \\
&(t_1 * t_2)[x := t'] = t_1[x := t'] * t_2[x := t'] \\
&(t_1 = t_2)[x := t'] = t_1[x := t'] = t_2[x := t'] \\
&(t_1 \leq t_2)[x := t'] = t_1[x := t'] \leq t_2[x := t'] \\
&(\neg t)[x := t'] = \neg t[x := t'] \\
&(t_1 \wedge t_2)[x := t'] = t_1[x := t'] \wedge t_2[x := t'] \\
&(t_1 \vee t_2)[x := t'] = t_1[x := t'] \vee t_2[x := t'] \\
&(t_1 t_2)[x := t'] = t_1[x := t']\, t_2[x := t'] \\
&(\lambda x.t)[x := t'] = \lambda x.t \\
&(\lambda y.t)[x := t'] = \lambda y.(t[x := t']) \quad \text{if } x \neq y \text{ and } y \notin \mathrm{FV}(t')
\end{aligned}
$$

Please note that for integer and boolean constants, substitution is not applied since these terms contain no free variables.

We can also introduce substitution, σ, as a function from variables to terms. Formally, $\sigma(t)$ is defined as:

$$\sigma(t) = t[\sigma(x_1)/x_1, \ldots, \sigma(x_n)/x_n] \quad \text{if the set of free variables of } t \text{ is } \{x_1, \ldots, x_n\}.$$

2.1.5 A Typing Algorithm

Our implementation checks that a given term is typeable using an algorithm that computes the most general type. The algorithm will compute a type A (with variables) for a given closed term t so that if $\Gamma \vdash t : B$ can be derived, then B can be obtained from A by replacing the variables in A by some types. For example, for the term $\lambda x.x$ the algorithm will compute the type $a \to a$ where a is a type variable. The two types $Bool \to Bool$ and $Int \to Int$ can be obtained by replacing a with $Bool$ and Int, respectively.

To compute the types of terms, we have used Martelli & Montanari's unification algoithm [11] which is the updated version of Robinson's unification algorithm [17, 16]. Robinson's original algorithm, while foundational, was known for its worst-case exponential time and space complexity. Martelli & Montanari's improvements addressed some of these challenges, making the algorithm more efficient and practical for determining type assignments in our context.

Unification Algorithm

The algorithm, commonly attributed to Martelli and Montanari [11], outlines the process of unification for a finite set $G = \{s_1 \doteq t_1, ..., s_n \doteq t_n\}$ of potential equations. The goal is to transform G into an equivalent set of equations of the form $\{x_1 \doteq u_1, ..., x_m \doteq u_m\}$, where $x_1, ..., x_m$ are distinct variables and $u_1, ..., u_m$ are terms containing none of the x_i. Such a set represents a substitution σ and we can say that $\sigma(s_i) = \sigma(t_i)$ for $1 \leq i \leq n$. If no solution exists, the algorithm terminates with $\perp$ (other authors use symbols like Ω or "fail" in this case). The algorithm operates with the following rules. In these rules, $\Rightarrow$ denotes the application of a rule, and $\perp$ signifies failure to find a unification. The algorithm proceeds by applying these rules iteratively until no further transformations are possible or a conflict arises.

1. **Identity Rule (Delete):**
$$G \cup \{t \doteq t\} \Rightarrow G$$
If an equation $t \doteq t$ appears in the set G, it can be safely removed because it does not introduce any new information.

2. **Decompose Rule:**
$$G \cup \{f(s_0, \ldots, s_k) \doteq f(t_0, \ldots, t_k)\} \Rightarrow G \cup \{s_0 \doteq t_0, \ldots, s_k \doteq t_k\}$$
If equations involve the same function symbol f with corresponding arguments, the equation is decomposed into individual equations for each argument.

3. **Conflict Rule:**
$$G \cup \{f(s_0, \ldots, s_k) \doteq g(t_0, \ldots, t_m)\} \Rightarrow \perp$$
If equations involve different function symbols f and g, or if they have different numbers of arguments k and m, it results in a conflict (failure to unify).

4. **Swap Rule:**
$$G \cup \{f(s_0, \ldots, s_k) \doteq x\} \Rightarrow G \cup \{x \doteq f(s_0, \ldots, s_k)\}$$
This rule allows swapping the position of a variable x and a term $f(s_0, \ldots, s_k)$ in an equation to facilitate unification.

5. **Eliminate Rule:**

$$G \cup \{x \doteq t\} \Rightarrow G\{x \mapsto t\} \cup \{x \doteq t\} \quad \text{if } x \notin \text{vars}(t) \text{ and } x \in \text{vars}(G)$$

   When a variable x does not appear in the term t and x is present in other equations in G, x can be eliminated by substituting t for x throughout G.

6. **Check Rule:**

$$G \cup \{x \doteq f(s_0, \ldots, s_k)\} \Rightarrow \perp \quad \text{if } x \in \text{vars}(f(s_0, \ldots, s_k))$$

   If a variable x appears in the term $f(s_0, \ldots, s_k)$, unification fails because it indicates a cyclic or conflicting dependency.

Unification Applied in the System

For each term, we compute the type unifying according to the rules stated below. If the unification is successful based on the unification algorithm described above, necessary substitution occurs in the term to compute the final type that we will see in the abstraction example below.

1. **Variable:** For each variable, we always compute a random general type. For example, for a variable x, the algorithm will compute the type a.

2. **NamedTerm:** For each NamedTerm, the type is simply the type specified in its declaration. For example, in the declaration:

$$succ :: Int \to Int = \lambda n.n + 1$$

   the type for the NamedTerm, $succ$ is $Int \to Int$.

3. **Integer Constants:** Always compute an Int type.

4. **Boolean Constants:** Always compute a $Bool$ type.

5. **Integer Operations:** Unify both the left and right terms' types with the Int type. If the unification succeeds, then return the final type, the Int type.

6. **Comparisons:** Unify both the left and right terms' types with the Int type. If the unification succeeds, then return the final type, the $Bool$ type.

7. **Not:** Unify the term's type with a $Bool$ type.

8. **Boolean Operations:** Unify both the left and right terms' types with the $Bool$ type. If the unification succeeds, then return the final type, the $Bool$ type.

9. **Conditional:** First of all, unify the If term with a $Bool$ type. If successful, then unify the Else and Then terms. If successful, the final type, the type of the Else or Then term, is returned.

10. **Abstraction and Recursion:** For the variable part, we compute the type according to the first rule. Then we compute the type of the body. Finally, we determine the function type of the variable and the body and return it as the final type. For example, consider the term $\lambda x.x + 1$. First, we compute a random type (a) for the variable x ($x \mapsto a$). According to the integer operations rule (rule 4) of our algorithm, we unify the left part (x) and the right part (1) of the body ($x + 1$) with the Int type. After unification, we get $a \mapsto Int$ and after substitution of the term, we get $x \mapsto Int$. Therefore, we can say that x is of type Int. Hence, the final type is $Int \rightarrow Int$.

11. **Application:** If the left term's type is a type variable, our algorithm creates another random type variable and computes a function type using the new and the existing type variables. Then, with the right term's type, we try unifying it with the left part of the function type. If successful, we return the right part of the function type as the final type. For example, consider the term $(x)y$. The left term, x has a type variable (a) according to rule 1, denoted as $x \mapsto$ a. So we create a function type $a \rightarrow b$. For the right term y, we compute another type variable (c). According to the algorithm, we unify (a) with (c). If the unification is successful, the right part (b) of the function type is returned. Hence, the final computed type for the term $(x)y$ is (b).

Find the type of a recursive term

Figure 2.1 shows the step-by-step procedure to find the type of a recursive term: (rec $f.\lambda n.$ if $n = 0$ then 1 else $n \times f(n-1)$) which is to be applied to an Int constant, 5. So, the term is, (rec $f.\lambda n.$ if $n = 0$ then 1 else $n \times f(n-1))5$. As the factorial of 5 is 120, which is an integer constant, we should therefore get the type of this term an Int type. Now, to understand how this program outputs an Int type, we break it down into smaller subprograms, find their types, and then finally come to the final result, Int. The steps are described in the following: -

1. The term is an application having its left and right terms. As shown in the figure 2.1, first, we check the left term which is a variable. For every variable in the system, we are

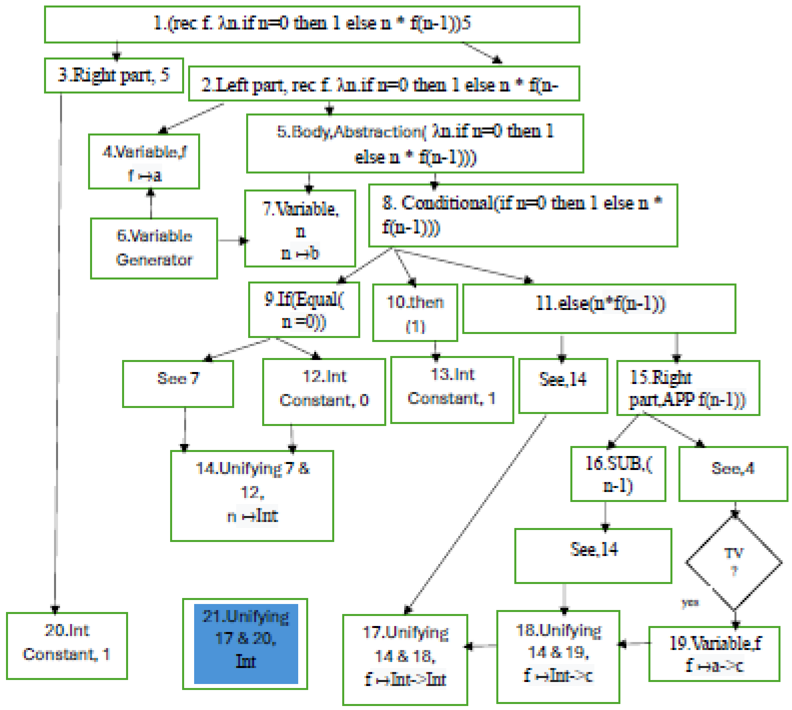

Figure 2.1: Typing of a PCFTerm.

generating a random type ($f \mapsto$ a) using the Variable generator (step 6) method. The right term, the abstraction (step 5), similarly has the variable and the body part.

2. The body of the abstraction (step 8) is a conditional term and has 3 parts (9, 10, 11) as shown in the figure. This breakdown continues until step 14 where we are unifying a type variable and an Integer constant following the unification algorithm[11]. After the substitution, we get $n \mapsto Int$

3. At step 15, there is an application operation, and at the decision-making step, we are checking whether the variable f is a TV (Type Variable). If so, we are creating a function type $f \mapsto (a \to c)$ (step 19) as mentioned in the Type section.

4. Step 17 shows the type of the left part of our program, and step 20 shows the type of the right part. Finally, at step 21, we have got our final type, Int, by applying Robson's unification substitution algorithm.

2.1.6 Semantics of PCF$^+$

We first need to define the notion of a normal form of a PCFTerm (Section 2.1.2). A normal form is the result of a computation, i.e., a PCFTerm that cannot be reduced further. Normal forms are defined by the type of the PCFTerm. First of all, a variable is a normal form of every type. In addition, we have the following normal forms:

- The PCFTerms of the form $\underline{n}$: Int, i.e., the integer constants, are normal forms of type Int.
- The PCFTerms True : Bool and False : Bool are normal forms of type Bool.
- The PCFTerms $\lambda x.t : \sigma \to \tau$ where t is an arbitrary term are normal forms of type $\sigma \to \tau$.

The semantics of PCF$^+$ is given by a relation $\to$ between PCFTerms and normal forms. The intuitive meaning of $p \to q$ is that the PCFTerm p evaluates to the normal form q. This relation is defined by rules, similar to the typing rules, as shown in Table 2.2.

From Table 2.3, we can better realize the PCFTerm semantics according to the rules discussed in Table 2.2.

2.1.7 Typing of Programs

As discussed above, declarations are the fundamental building blocks of a program, and each declaration includes a $PCFTerm$ as its body, which is one of its three key components. To determine the type of a program, we primarily need to ascertain the types of all the declarations. The steps to follow are outlined below:

1. First, for each declaration, extract the body (which is a $PCFTerm$) and compute its type using the typing algorithm (Section 2.1.5) and the typing rules (Section 2.1.4).
2. Next, unify the computed type with the declared type of the declaration by applying the unification algorithm (Section 2.1.5).
3. Then, examine the declaration named $main$ and verify if it is of type $Bool$ or Int. If $main$ is of neither type, the system raises an exception: "main is not of type Int or Bool".
4. Finally, if the unification is successful, the type of the program is the type of the declaration named $main$.

2.1.8 Execution of Programs

Our PCF$^+$ program execution always starts with the declaration named 'main'. We simply take the 'body' of 'main' and then execute it. The execution procedures follows the same rules for a PCFTerms as detailed in Table 2.2. The execution result must be either a boolean value ('True' or 'False') or an integer constant.

To better understand how the execution of a program proceeds, refer to table 2.4.

Table 2.4: Step-by-step execution of a PCF$^+$ program.

PCFTerms/Programs	**Steps**
Initial Program	fact :: Int $\to$ Int = recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1)$; succ :: Int $\to$ Int = $\lambda n.\, n+1$; main :: Int = fact(succ(1));
NamedTerm(main)	fact(succ(1))
NamedTerm(fact)	recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1)$ (succ(1))
Recursion	$\lambda n.$ if $n = 0$ then 1 else $n *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ $(n-1)$ (succ(1))
Application	if (succ(1)) $=$ 0 then 1 else (succ(1)) $*$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ (succ$(1)-1)$
NamedTerm(succ)	if $((\lambda n.\, n{+}1)\, 1) = 0$ then 1 else (succ(1))$*$(recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ (succ$(1)-1)$
Application	if $(1 + 1 = 0)$ then 1 else (succ(1)) $*$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ (succ$(1)-1)$
Addition	if $(2 = 0)$ then 1 else (succ(1)) $*$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ (succ$(1)-1)$
Comparison	if False then 1 else (succ(1)) $*$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ (succ$(1)-1)$
Conditional	(succ(1)) $*$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ (succ$(1)-1)$
NamedTerm(succ)	$(\lambda n.\, n+1)\, 1 *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ (succ$(1)-1)$
Application	$(1{+}1){*}$(recf. $\lambda n.$ if $n = 0$ then 1 else $n{*}f(n{-}1))$ (succ$(1){-}1)$
Addition	$2 *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ (succ$(1)-1)$
Recursion	$2 * \lambda n.$ if $n = 0$ then 1 else $n *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ $(n-1)$ (succ$(1)-1)$

Application	$2 * \text{if } (\text{succ}(1) - 1) = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
NamedTerm(succ)	$2 * \text{if } ((\lambda n. \, n + 1) \, 1 - 1) = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Application	$2 * \text{if } (1 + 1 - 1) = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Addition	$2 * \text{if } (2 - 1) = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Subtraction	$2 * \text{if } 1 = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Comparison	$2 * \text{if False then } 1 \text{ else } (\text{succ}(1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Conditional	$2 * (\text{succ}(1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
NamedTerm(succ)	$2 * ((\lambda n. \, n + 1)(1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Application	$2 * ((1 + 1) - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Addition	$2 * (2 - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Subtraction	$2 * (1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (\text{succ}(1) - 1 - 1)$
Recursion	$2 * (1) * (\lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, (n-1)) \, (\text{succ}(1) - 1 - 1)$
Application	$2 * (1) * (\lambda n. \text{ if } (\text{succ}(1) - 1 - 1) = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1 - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, ((\text{succ}(1) - 1 - 1) - 1))$
Namedterm(succ)	$2 * (1) * (\lambda n. \text{ if } ((\lambda n. \, n + 1)(1) - 1 - 1) = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1 - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, ((\text{succ}(1) - 1 - 1) - 1))$
Application	$2 * (1) * (\lambda n. \text{ if } (((1+1) - 1 - 1) = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1 - 1) * (\text{recf. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) \, ((\text{succ}(1) - 1 - 1) - 1))$

Addition	$2 * (1) * (\lambda n.\, \text{if } ((2-1-1) = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1 - 1) * (\text{rec} f.\, \lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) ((\text{succ}(1) - 1 - 1) - 1))$
Subtraction	$2 * (1) * (\lambda n.\, \text{if } (0 = 0 \text{ then } 1 \text{ else } (\text{succ}(1) - 1 - 1) * (\text{rec} f.\, \lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) ((\text{succ}(1) - 1 - 1) - 1))$
Comparison	$2 * (1) * (\lambda n.\, \text{if } (True \text{ then } 1 \text{ else } (\text{succ}(1) - 1 - 1) * (\text{rec} f.\, \lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) ((\text{succ}(1) - 1 - 1) - 1))$
Conditional	$2 * (1) * (1)$
Multiplication	2

Table 2.2: Semantics of PCF$^+$

Name	**Rules**
Variables	$x \to x$ if x is a variable
Integer constants	$\underline{n} \to \underline{n}$ for all integers n
Boolean constants	True $\to$ True and False $\to$ False
Integer operations	$\dfrac{t_1 \to \underline{n_1} \quad t_2 \to \underline{n_2}}{t_1 \oplus t_2 \to \underline{n_1 \oplus n_2}}$ $\oplus \in \{+, *, -\}$
Comparisons	$\dfrac{t_1 \to b_1 \quad t_2 \to b_2}{t_1 \oplus t_2 \to b}$ $\oplus \in \{\leq, =\}$ (if $b_1 \oplus b_2 = b$)
Not operation	$\dfrac{t \to b}{\neg t \to b'}$ (b' is the logical negation of b)
Boolean operations	$\dfrac{t_1 \to b_1 \quad t_2 \to b_2}{t_1 \oplus t_2 \to b}$ $\oplus \in \{\vee, \wedge\}$ (if $b_1 \oplus b_2 = b$)
Conditional	$\dfrac{t_1 \to \text{True} \quad t_2 \to n}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \to n}$ $\dfrac{t_1 \to \text{False} \quad t_3 \to n}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \to n}$
Abstraction	$\lambda x.t \to \lambda x.t$
Recursion	$\dfrac{t[\text{rec } x.t/x] \to n}{\text{rec } x.t \to n}$
Application	$\dfrac{t_1 \to \lambda x.t_1' \quad t_1'[t_2/x] \to n}{t_1 t_2 \to n}$

Table 2.3: Detailed Computation Steps for a PCFTerm

PCFTerm	**steps**
Initial PCFTerm	(recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1)$) 3
Recursion	($\lambda n.$ if $n = 0$ then 1 else $n *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(n-1))$ 3
Conditional	if $3 = 0$ then 1 else $3 *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(3-1)$
Comparison	if $False$ then 1 else $3 *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(3-1)$
Conditional	$3 *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(3-1)$
Recursion	$3 *$ ($\lambda n.$ if $n = 0$ then 1 else $n *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(n-1))(3-1)$
Application	$3 *$ (if $(3-1) = 0$ then 1 else $(3-1) *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))((3-1)-1))$
Subtraction	$3 *$ (if $2 = 0$ then 1 else $(3-1) *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))((3-1)-1))$
Comparison	$3 *$ (if $False$ then 1 else $(3-1) *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))((3-1)-1))$
Conditional	$3 * ((3-1) *$(recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))((3-1)-1))$
Subtraction	$3 * (2 *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))((3-1)-1))$
Recursion	$3 * (2 *$ ($\lambda n.$ if $n = 0$ then 1 else $n *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(n-1))((3-1)-1))$
Application	$3 * (2 *$ (if $((3-1)-1) = 0$ then 1 else $((3-1)-1) *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(((3-1)-1)-1)))$
Subtraction	$3 * (2 *$ (if $1 = 0$ then 1 else $1 *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(((3-1)-1)-1)$
Comparison	$3 * (2 *$ (if $False$ then 1 else $1 *$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(((3-1)-1)-1))$
Conditional	$3 * (2 * (1 *$(recf. $\lambda n.$ if $n = 0$ then 1 else $n * f(n-1))(((3-1)-1)-1))$
Recursion	$3 \times (2 \times (1 \times$ ($\lambda n.$ if $n = 0$ then 1 else $n \times$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n \times f(n-1))(((3-1)-1)-1))))$
Application	$3 \times (2 \times (1 \times$ (if $(((3-1)-1)-1) = 0$ then 1 else $(((3-1)-1)-1) \times$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n \times f(n-1))(n-1))))$
Subtraction	$3 \times (2 \times (1 \times$ (if $0 = 0$ then 1 else $(((3-1)-1)-1) \times$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n \times f(n-1))(n-1))))$
Comparison	$3 \times (2 \times (1 \times$ (if $True$ then 1 else $(((3-1)-1)-1) \times$ (recf. $\lambda n.$ if $n = 0$ then 1 else $n \times f(n-1))(n-1))))$
Conditional	$3 \times (2 \times (1 \times 1))$
Multiplication	6

Chapter 3

Combinators and Translation

Combinators are higher-order functions that define results through function application alone, simplifying the representation of computations. Combinatory logic, introduced by Moses Schönfinkel and Haskell Curry, eliminates the need for variables in mathematical logic by using combinators. They are particularly beneficial in functional programming languages, where the absence of variables leads to simpler and often more efficient code.In programming computable functions (PCF), combinators play a crucial role by streamlining computations and function definitions. Lambda expressions, core to functional programming languages, enable function definition and application through variable abstraction. These expressions can be systematically translated into combinator expressions using predefined rules (Section 3.2), demonstrating the completeness of the S-K basis in combinatory logic. This means any function expressible in lambda calculus can also be represented using a limited set of primitive combinators. Consequently, all operations in our PCF$^+$ programs (Section 2.1.3) are translated into combinator programs (Section 3.1.2) as binary trees, whose leaves are one of our combinators discussed in the following sections. This elimination or reduction process of PCF$^+$ programs, highlights the theoretical robustness and practical simplicity of combinatory logic in representing computable functions, making it a powerful tool in both mathematical logic and computer science.

For the translation of our PCF$^+$ programs, we have integrated a diverse array of combinators to ensure comprehensive and efficient term conversion that will be discussed in the following section.

3.1 Combinators

Every program in PCF can be translated into its equivalent combinator program (see Section 3.1.2). As discussed in Chapter 2, just as PCF terms are the basic building blocks of programs, combinator terms similarly form the fundamental building blocks of combinator programs.

Both combinator terms and combinator programs will be discussed in the following section.

3.1.1 Combinator Terms

A combinator term has one of the following forms. The primitive functions mentioned here, are functions that contain no free variables when expressed as lambda terms. Our newly introduced combinators include INTEGER, NTERM, TRUE, FALSE, ADD, SUB, MULT, EQUAL, LEQUAL, NOT, AND, OR, and CONDITIONAL. To translate the recursive terms, we have incorporated the fixed-point combinator, also known as the Y combinator. All combinators can be categorized into two groups: call-by-name and call-by-value evaluation strategies, which define how arguments are handled in the PCF^{+} system. S, K, I and Y uses call-by-name strategies, meaning we simply execute the rule without executing the parameters first. Also, Y can be expressed in terms of SKI but we use it explicitly for efficiency reasons. On the contrary, the other rules as mentioned above uses call-by-value method, meaning we have to execute the parameters first before we can apply the rule. This is different from SKI and Y where we execute the parameters first other than parameters.

Table 3.1: Combinatory Term Forms

Syntax	**Name**	**Description**
x	Variable	A character or string representing a combinatory term.
P	Primitive function	One of the combinator symbols I, K, S.
(M N)	Application	Applying a function to an argument. M and N are combinatory terms.

The behavior of each combinator is defined as follows:-

call-by-name rules

Informally, in programming language terms, a tree $(t_1 t_2)$ can be viewed as a function t_1 applied to an argument t_2. When evaluated, the function is applied to the argument, and the tree transforms into another tree, effectively ‘returning a value.’ The ‘function,’ ‘argument,’ and ‘value’ are either combinators or binary trees, which can also be considered functions if needed.

- **I Term:**

$$It \to t$$

Returns its argument.

- **K Term:**

$$Kt_1t_2 \to t_1$$

When K is applied to an argument t_1, it produces a one-argument constant function Kt_1, which, when applied to any argument t_2, returns t_1.

- **S Term:**

$$St_1t_2t_3 \to t_1t_3(t_2t_3)$$

S is a substitution operator that takes three arguments. It applies the first argument to the third which is then applied to the result of the second argument applied to the third.

- **Y Term:**

$$Y\ t\ =\ t\,(Y\ t)$$

The Y combinator (or fixpoint combinator) is a higher-order function that, when given a function t, returns a fixed point of that function. In other words, applying t to $Y\ t$ results in $Y\ t$ itself. Formally, if Y is a fixed-point combinator and t is a function with one or more fixed points, then Y t is one of these fixed points. We have used Y combinators for the translation of our PCF$^+$ recursive terms.

call-by-value rules

As mentioned above, we are executing the parameter first before applying the rule.

- **Integer Term:**

$$\underline{n} \to \underline{n} \quad \text{for all integers n}$$

- **Named Term:**

$$\text{NamedTerm} \to \text{NTERM}$$

- **TRUE Term:**

$$\text{True} \rightarrow \text{TRUE}$$

- **FALSE Term:**

$$\text{False} \rightarrow \text{FALSE}$$

- **ADD Term:**

$$\frac{t_1 \rightarrow \underline{n_1} \quad t_2 \rightarrow \underline{n_2}}{\text{ADD } \ t_1 t_2 \rightarrow \underline{n_1 + n_2}}$$

- **SUB Term:**

$$\frac{t_1 \rightarrow \underline{n_1} \quad t_2 \rightarrow \underline{n_2}}{\text{SUB } \ t_1 t_2 \rightarrow \underline{n_1 - n_2}}$$

- **MULT Term:**

$$\frac{t_1 \rightarrow \underline{n_1} \quad t_2 \rightarrow \underline{n_2}}{\text{MULT } \ t_1 t_2 \rightarrow \underline{n_1 * n_2}}$$

- **EQUAL Term**

$$\frac{t_1 \rightarrow b_1 \quad t_2 \rightarrow b_2}{\text{EQUAL } t_1 t_2 \rightarrow b} \quad (\text{if } (b_1 = b_2) = b)$$

- **LEQUAL Term**

$$\frac{t_1 \rightarrow b_1 \quad t_2 \rightarrow b_2}{\text{LEQUAL } t_1 t_2 \rightarrow b} \quad (\text{if } (b_1 = b_2) = b)$$

- **NOT Term:**

$$\frac{t \rightarrow b}{\text{NOT } t \rightarrow b'} \quad (b' \text{ is the logical negation of } \ b)$$

- **AND Term**

$$\frac{t_1 \rightarrow b_1 \quad t_2 \rightarrow b_2}{\text{AND } t_1 t_2 \rightarrow b} \quad (\text{if } b_1 \wedge \ b_2 = b)$$

- **OR Term**

$$\frac{t_1 \rightarrow b_1 \quad t_2 \rightarrow b_2}{\text{OR } t_1 t_2 \rightarrow b} \quad (\text{if } b_1 \vee b_2 = b)$$

- **CONDITIONAL Term**

$$\frac{t_1 \rightarrow \text{True} \quad t_2 \rightarrow n}{\text{IF } t_1 t_2 t_3 \rightarrow n} \qquad \frac{t_1 \rightarrow \text{False} \quad t_3 \rightarrow n}{\text{IF } t_1 t_2 t_3 \rightarrow n}$$

3.1.2 Combinator programs

A combinator program is essentially a translated version of a PCF program. For instance, the combinator version of the program shown in Equation 2.1 is provided below. As illustrated in Equation 3.1, for every declaration (see Section 2.1.3), PCF terms are translated into their equivalent combinator terms, following the specified translation rules (see Section 3.2).

$$\begin{aligned}
\texttt{fact} &= Y\,(S\,(K\,(S\,(S\,(S\,(K\,\texttt{IF})\,(S\,(S\,(K\,\texttt{EQUAL})\,I)\,(K\,0)))\,(K\,1))))\\
&\quad (S\,(K\,(S\,(S\,(K\,\texttt{MULT})\,I)))\,(S\,(S\,(K\,S)\,(S\,(K\,K)\,I))\,(K\,(S\,(S\,(K\,\texttt{SUB})\,I)\,(K\,1))))))\\
\texttt{succ} &= S\,(S\,(K\,\texttt{ADD})\,I)\,(K\,1)\\
\texttt{main} &= \texttt{fact}\,(\texttt{succ}\,4)
\end{aligned} \tag{3.1}$$

3.2 Translation

With the combination of different combinators discussed in 3.1.1, we can produce combinators that are extensionally equivalent to any PCF$^+$ terms and, according to Church's thesis, to any computable function. This is shown through a translation, $T[\quad]$, which converts any lambda term into an equivalent combinator. Translation rules, $T[\quad]$ can be defined as following:-

1. $T[x] \Rightarrow x$
2. $T[(t_1 t_2)] \Rightarrow (T[t_1]T[t_2])$
3. $T[\lambda x.t] \Rightarrow (K\ T[t])$ (if x does not occur free in t)
4. $T[\lambda x.x] \Rightarrow I$

5. $T[\lambda x.\lambda y.t] \Rightarrow T[\lambda x.T[\lambda y.t]]$ (if x occurs free in t)
6. $T[\lambda x.(t_1 t_2)] \Rightarrow (S\, T[\lambda x.t_1]\; T[\lambda x.t_2])$ (if x occurs free in t_1 or t_2)

Please note that the translation rules listed above are not a well-typed mathematical function but rather a term rewriter. Although it eventually yields a combinator, the transformation may generate intermediate expressions that are neither PCF$^+$ terms nor combinators, particularly via rule (5). To resolve this issue, we convert each PCF$^+$ term to intermediate terms before translating them into combinator terms. The intermediate rules are defined as follows:

1. $T[NamedTerm] \Rightarrow NTERM$
2. $T[TRUE] \Rightarrow TRUE$
3. $T[FALSE] \Rightarrow FALSE$
4. $T[n] \Rightarrow n$ (for any integer combinator, n)
5. $T[t_1 + t_2] \Rightarrow$ ADD $[t_1]$ $[t_2]$
6. $T[t_1 - t_2] \Rightarrow$ SUB $[t_1]$ $[t_2]$
7. $T[t_1 * t_2] \Rightarrow$ MULT $[t_1]$ $[t_2]$
8. $T[t_1 = t_2] \Rightarrow$ EQUAL $[t_1]$ $[t_2]$
9. $T[[t_1] \leq [t_2]] \Rightarrow$ LEQUAL $[t_1]$ $[t_2]$
10. $T[$not $[t]] \Rightarrow$ NOT $[t]$
11. $T[[t_1]$ and $[t_2]] \Rightarrow$ AND $[t_1]$ $[t_2]$
12. $T[[t_1]$ or $[t_2]] \Rightarrow$ OR $[t_1]$ $[t_2]$
13. $T[$IF $[t_1]$ THEN $[t_2]$ ELSE $[t_3]] \Rightarrow$ IF $[t_1]$ $[t_2]$ $[t3]$
14. $T[$rec $x.t] \Rightarrow (Y\; T[\lambda x.t])$

3.2.1 Example

From the following example, we can prove that the original reduction of a PCFTerm and the reduction after translating into combinators give the same result.

PCF$^+$ Program: $(\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } 2)\ 1$
Intermediate Term: $(\lambda n.\text{IF (EQUAL } n\ 0)\ 1\ 2)\ 1$
Now translating this gives:

$$= T[\lambda n.\text{IF (EQUAL } n\ 0)\ 1\ 2]\ T[1] \quad \text{(rule 2)}$$
$$= ST[\lambda n.\text{IF (EQUAL } n\ 0)\ 1]\ T[\lambda n.2]\ 1 \quad \text{(rule 6)}$$
$$= S(ST[\lambda n.\text{IF (EQUAL } n\ 0)]\ T[\lambda n.1])\ (K2)\ 1 \quad \text{(rule 6,3)}$$
$$= S(S(ST[\lambda n.\text{IF}]\ T[\lambda n.\text{EQUAL } n\ 0])\ (K1))\ (K2)\ 1 \quad \text{(rule 6,3)}$$
$$= S(S(S(K\ \text{IF})\ (ST[\lambda n.\text{EQUAL } n]\ T[\lambda n.0]))\ (K1))\ (K2)\ 1 \quad \text{(rule 3,6)}$$
$$= S(S(S(K\ \text{IF})\ (S(ST[\lambda n.\text{EQUAL}]\ T[\lambda n.n])\ (K0)))\ (K1))\ (K2)\ 1 \quad \text{(rule 6,3)}$$
$$= S(S(S(K\ \text{IF})\ (S(S(K\ \text{EQUAL})\ I)\ (K0)))\ (K1))\ (K2)\ 1 \quad \text{(rule 3)}$$

If we reduce this combinator term, we indeed get the correct result:

$$\to S(S(S(K\ \text{IF})\ (S(S(K\ \text{EQUAL})\ I)\ (K0)))\ (K1))\ (K2)\ 1$$
$$\to S(S(K\ \text{IF})\ (S(S(K\ \text{EQUAL})\ I)\ (K0)))\ (K1)\ 1\ (K2\ 1) \quad \text{(outermost S)}$$
$$\to S(K\ \text{IF})\ (S(S(K\ \text{EQUAL})\ I)\ (K0))\ 1\ (K1\ 1)\ (K2\ 1) \quad \text{(outermost S)}$$
$$\to K\ \text{IF}\ 1\ (S(S(K\ \text{EQUAL})\ I)\ (K0)\ 1)\ (K1\ 1)\ (K2\ 1) \quad \text{(outermost S)}$$
$$\to \text{IF}\ (S(S(K\ \text{EQUAL})\ I)\ (K0)\ 1)\ (K1\ 1)\ (K2\ 1) \quad \text{(outermost K)}$$
$$\to \text{IF}\ (S(K\ \text{EQUAL})\ I\ 1\ (K0\ 1))\ (K1\ 1)\ (K2\ 1) \quad \text{(outermost S in condition of IF)}$$
$$\to \text{IF}\ (K\ \text{EQUAL}\ 1\ (I\ 1)\ (K0\ 1))\ (K1\ 1)\ (K2\ 1) \quad \text{(outermost S in condition of IF)}$$
$$\to \text{IF (EQUAL}\ (I\ 1)\ (K0\ 1))\ (K1\ 1)\ (K2\ 1) \quad \text{(outermost K in condition of IF)}$$
$$\to \text{IF (EQUAL}\ 1\ (K0\ 1))\ (K1\ 1)\ (K2\ 1) \quad \text{(I in first parameter of EQUAL)}$$
$$\to \text{IF (EQUAL}\ 1\ 0)\ (K1\ 1)\ (K2\ 1) \quad \text{(K in second parameter of EQUAL)}$$
$$\to \text{IF}\ FALSE\ (K1\ 1)\ (K2\ 1) \quad \text{(EQUAL)}$$
$$\to K\ 2\ 1 \quad \text{(IF)}$$
$$\to 2 \quad \text{(K)}$$

Original Reduction:

$\rightarrow (\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } 2)\ 1 \quad \text{(Application)}$

$\rightarrow \text{if } 1 = 0 \text{ then } 1 \text{ else } 2 \quad \text{(Substitution due to left part being an Abstraction term)}$

$\rightarrow 1 = 0 \quad \text{(If part of the conditional term is an Equal term)}$

$\rightarrow 2 \quad \text{(Else term)}$

As we can see above, during the reduction from the original PCF$^+$ term, initially, it is an application. In the first step, we evaluate the left part, which is an abstraction term. According to the rules, we perform substitution into the abstraction term with the right part of the application. After the substitution, the resulting term is a conditional term. Next, we evaluate the conditional term starting with the `IF` part. The `IF` condition, an `Equal` term (`(1 = 0)`), is evaluated, and since `Equal` returns `False`, the `Else` part is executed, resulting in the value `2`. This final result, `2`, matches exactly with the result obtained from the combinator translation after reduction.

Chapter 4

An Abstract Machine

Our combinator terms, a translated version of a program, can also be visualized as trees (as shown in Figure 4.1) and must be encoded in binary or byte format to be interpretable by machines. This binary encoding allows the combinator terms to be efficiently stored and processed within a computer's memory and execution systems. An Abstract machine basically executes our programs. It has mainly two components i.e the heap (Section 4.1) and the stack (Section 4.2). Programs are stored in the heap and the stack facilitates the manipulation of various operations during the program execution.

In this chapter, we will explore the techniques for storing programs in the memory and the implementation of an abstract machine to execute these programs. We have developed two implementation of this abstract machine: one in Java, which operates on a byte array using a push-back stack mechanism, and another in x86 machine code, where the original program is stored under a specific label, 'prog'. The entire process is divided into two key steps: storage and program execution (i.e., using Java and x86 machine code).

4.1 Storage (heap)

Similar to a machine language, where every instruction has an operational code (op code), we need to define op codes for all of our combinators, as listed in table 3.1.1. Some combinators will have parameters, akin to regular machine instructions. For instance, the LOAD instruction includes two parameters: the op code, and an additional parameter specifying where to store the value.

Among our combinators, four will have parameters: Application, NamedTerm, Integer, and Boolean Constants. For the Application combinator, it actually has two parameters, but we only need to list one parameter, which is the address of the right subtree. We do not need the address of the left subtree, as it is simply the subsequent entry in the storage.

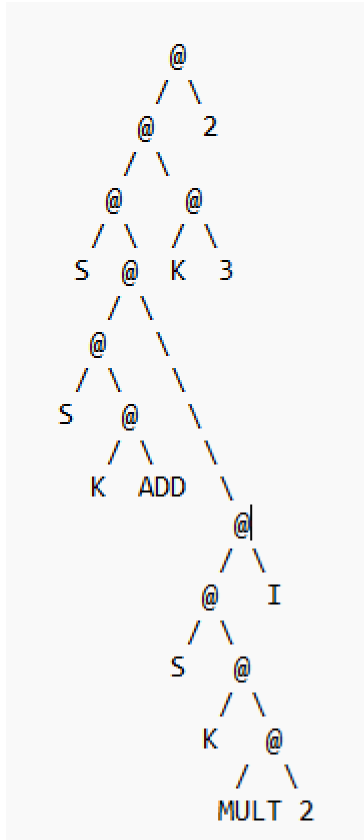

Figure 4.1: Graph representation of $(S(S(K\ ADD)(S(K(MULT\ 2))I))(K\ 3)2$

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td></tr>
<tr><td>128</td><td colspan="4">9</td><td colspan="4">118</td><td>128</td><td colspan="4">18</td><td colspan="4">103</td><td>128</td><td colspan="4">27</td><td colspan="4">28</td><td>131</td><td>128</td><td></td></tr>
<tr><td>30</td><td>31</td><td>32</td><td>33</td><td>34</td><td>35</td><td>36</td><td>37</td><td>38</td><td>39</td><td>40</td><td>41</td><td>42</td><td>43</td><td>44</td><td>45</td><td>46</td><td>47</td><td>48</td><td>49</td><td>50</td><td>51</td><td>52</td><td>53</td><td>54</td><td>55</td><td>56</td><td>57</td><td>58</td><td>59</td></tr>
<tr><td colspan="3">37</td><td colspan="4">58</td><td>128</td><td colspan="4">46</td><td colspan="4">47</td><td>131</td><td>128</td><td colspan="4">56</td><td colspan="4">57</td><td>130</td><td>64</td><td>128</td><td></td></tr>
<tr><td>60</td><td>61</td><td>62</td><td>63</td><td>64</td><td>65</td><td>66</td><td>67</td><td>68</td><td>69</td><td>70</td><td>71</td><td>72</td><td>73</td><td>74</td><td>75</td><td>76</td><td>77</td><td>78</td><td>79</td><td>80</td><td>81</td><td>82</td><td>83</td><td>84</td><td>85</td><td>86</td><td>87</td><td>88</td><td>89</td></tr>
<tr><td colspan="3">67</td><td colspan="4">102</td><td>128</td><td colspan="4">76</td><td colspan="4">77</td><td>131</td><td>128</td><td colspan="4">86</td><td colspan="4">87</td><td>130</td><td>128</td><td colspan="2"></td></tr>
<tr><td>90</td><td>91</td><td>92</td><td>93</td><td>94</td><td>95</td><td>96</td><td>97</td><td>98</td><td>99</td><td>100</td><td>101</td><td>102</td><td>103</td><td>104</td><td>105</td><td>106</td><td>107</td><td>108</td><td>109</td><td>110</td><td>111</td><td>112</td><td>113</td><td>114</td><td>115</td><td>116</td><td>117</td><td>118</td><td>119</td></tr>
<tr><td colspan="2">96</td><td colspan="4">97</td><td>69</td><td>1</td><td colspan="4">2</td><td>7</td><td>128</td><td colspan="4">112</td><td colspan="4">113</td><td>130</td><td>1</td><td colspan="4">3</td><td>1</td><td></td></tr>
<tr><td>120</td><td>121</td><td>122</td></tr>
<tr><td colspan="3">2</td></tr>
</table>

Figure 4.2: Storage for the program [main :: Int $=$ $(\lambda n.\ 2 \cdot n + 3)\ 2$]

Table 4.1: Op Code for the Combinators

Combinator	OpCode	Comment
BOOL	0	
INT	1	
	2-63	empty, left for future constants
ADD	64	
AND	65	
COND	66	
EQ	67	
LEQ	68	
MUL	69	
NOT	70	
OR	71	
SUB	72	
	73-127	empty, left for future operations
APP	128	
I	129	
K	130	
S	131	
Y	132	
JMP	133	
	134-255	empty, left for future instructions

Both the NamedTerm and the Integer term use one parameter to store the NamedTerm and the integer value respectively. Boolean terms (True, False) use one byte as a parameter, either 0 or 1, to indicate True or False.

Please note, the JMP instruction is incorporated for NamedTerms, allowing for scenarios where multiple PCFTerms declared in a program can call each other. For example, in equation 2.1, the declaration of 'main' includes two NamedTerms: 'succ' and 'fact'. In this scenario, the JMP instruction translates to a jump to the memory address where the NamedTerm is stored.

All combinators with their respective op codes are listed in Table 4.1. Using these op codes, we can construct the storage for a specific combinator term.

For storing the combinators in the array of bytes or in the storage, we have used several methods that are discussed below in the subsequent subsections.

4.1.1 Accessing Memory Data

The following methods facilitate easy access to retrieving and storing data, as well as the allocation of new space in memory.

getByte(int addr): Given a memory address, this method returns the value stored at that address.

storeByte(int addr, byte value): This method sets the specified value at the given address in memory.

getInt(int addr): An integer consists of 4 bytes. Given the address of the first byte of the integer value, we retrieve all four bytes and then use `ByteBuffer` to wrap them into the original integer value.

storeInt(int addr, int value): This method is just the reverse process of retrieving an integer. It converts the given integer value into an array of bytes using `ByteBuffer` and then stores these bytes at the specified address in memory.

allocate(int numBytes): This method creates new spaces in memory equal to the number of `numBytes`.

4.1.2 Storing Combinators

We will now explore the process of storing each combinator in memory using the opCodes listed in Table 4.1 and the memory access methods discussed in Section 4.1.1.

BOOL: Use the `allocate` method to create 2 bytes in memory. Store the opCode `0` in the first byte, and either `0` or `1` in the second byte to represent `TRUE` or `FALSE`, respectively.

INT: Use the `allocate` method to create 5 bytes in memory. Store the opCode `1` in the first byte, and the integer in the next four byte using the storeInt() method in the memory.

APP: Use the `allocate` method to create 9 bytes in memory. Store the opCode `128` in the first byte. Using storeInt() method, set the address of the left subtree in bytes `5` through `8` by retrieving the value from position `addr` + 9 in the storage. The address of the right subtree is set in bytes `1` through `4` once the traversal of the right subtree is completed.

NTERM: Use the `allocate` method to create 5 bytes in memory. Store the opCode `133` in the first byte, and the address of the NTerm in the next four byte using the storeInt() method in the memory.

For the remaining combinators—ADD, AND, COND, EQ, LEQ, MUL, NOT, OR, SUB, I, K, S, and Y—we allocate only 1 byte in memory to store the corresponding op-

Code, as listed in the opCode table 4.1.

4.1.3 Storing [main :: Int $= (\lambda n.\ 2 \cdot n + 3)\ 2$]

We will now examine how the combinator program [main :: Int $= (\lambda n.\ 2 \cdot n + 3)\ 2$] is stored in memory, as detailed in figure 4.2, utilizing the op codes listed in Table 4.1. This program can also be represented graphically, as illustrated in Figure 4.1, which aids in understanding the process of storage creation. Respective figures and tables for the storage as discussed above are listed in 4.1, 4.2 and 4.1.

Program: [main :: Int $= (\lambda n.\ 2 \cdot n + 3)\ 2$]
Combinator program: $S(S(K\ ADD)(S(K(MULT\ 2))I))(K\ 3)2$

In Figure 4.1, the @ symbol indicates an Application, which has both left and right parameters, as discussed. Starting at the root, we encounter an application, so 128, the opcode value for Application as shown in Table 4.1, is inserted into memory. At the same time, 8 bytes are allocated in memory for the left and right parameters respectively. The first four bytes (1-4) store the address of the left subtree, which is again an application, and the remaining bytes (5-8) will store the address of the right subtree once its traversal is completed.

During storage creation, we traverse the graph in Depth-First Search (DFS) mode, meaning the right subtree will only be traversed after completing all left subtree traversals. Returning to our example, for the right subtree, an INT combinator 2, space is allocated only after it is traversed. As seen in Figure 4.2, 2 is traversed last (after all left traversals are completed) and then allocated 5 bytes (118-122), with 1 byte for the opcode and 4 bytes for storing the integer constant. Let's also examine the first S encountered when traversing the graph in Figure 4.1, with the space allocation starting at the 27th position in memory. At this point, as there are no further left subtrees to traverse, we immediately start traversing the right subtree, which is again an application. Space allocation for this begins at the 28th position in storage. Thus, we traverse, allocate, and update the space, resulting in the final storage as shown in Table 4.2.

4.2 Program Execution in Java

During the execution of the program, an address stack is introduced on top of the storage (see Figure 4.3). This address stack has a specified capacity, ensuring efficient management of addresses during execution. To facilitate this, we have implemented several methods for manipulating the address stack, along with two pointers, stackTop and backUpTop, to enable a push-back stack mechanism.

The address stack, in conjunction with the opCodes as listed in Table 4.1 and the byte array of storage shown in the Figure 4.2, allows for efficient execution and management of the program. This mechanism ensures that the program can correctly store and retrieve addresses, handle operations, and maintain the state of the computation effectively. The detailed techniques and methods used to implement this functionality are discussed in the following sections.

4.2.1 Address Stack

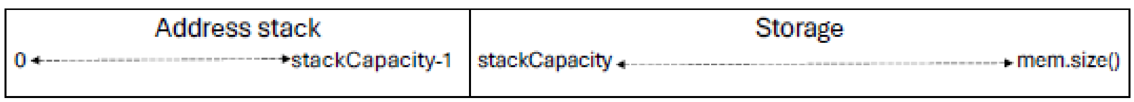

Figure 4.3: Storage during execution

We have implemented the address stack within the same memory where the program is stored to mirror the behavior of machine code environments such as WebAssembly. To achieve this, we allocated space at the top of the memory, sized according to the stack capacity. This means that the stack capacity defines the total capacity of the address stack.

To manage the address stack efficiently, we introduced two pointers: `stackTop` and `backUpTop`. These pointers indicate the current top positions of the address stack and the backup stack, respectively. Initially, `stackTop` is set to -4, and `backUpTop` is set to the stack capacity.

The address stack begins at address 0 and grows upwards as items are added, while the backup stack starts at the stack capacity and grows downwards. This configuration ensures that the address stack and backup stack can coexist within the allocated memory space without interfering with each other.

4.2.2 Execution procedures and steps

We have completed the process of creating storage for combinators. Once the storage is established, various manipulations are performed on the address stack and the array of bytes during execution. These manipulations ultimately produce the final output of the program. The mechanisms and techniques involved, along with the methods used, are detailed below:

Stack methods

push: Increases the `stackTop` by 4 and sets the value at this position using the `storeInt` method as discussed in the section 4.1.1.
pop: Retrieves the value at the `stackTop` position from the memory and then decreases the `stackTop` by 4.
peek: Retrieves the value at the `stackTop` position in the memory.
peekSecond: Retrieves the value at `stackTop - 4` in the memory.
peekThird: Retrieves the value at `stackTop - 8` in the memory.
backUp: Decreases `backUpTop` by 4, stores the value at the `stackTop` position in memory to the `backUpTop` position, and then decreases `stackTop` by 4.
restore: Reverses the `backUp` procedure. Increases `stackTop` by 4, stores the value at the `backUpTop` position in memory to the `stackTop` position, and then increases `backUpTop` by 4.
stackSize: Returns the stack size. Only if `backUpTop` equals `stackCapacity` and `stackTop` equals 0 at the same time, the stack size is 0.

To sum up, the allocated memory and the two pointers, $stackTop$ and $backUpTop$ are used to implement the above methods. Now, we will examine in detail how the allocated memory is manipulated and how the address stack is modified for each combinator. The subsequent section outlines the entire mechanism, providing a clear understanding of the steps involved in the process:

Manipulation

Starting at position 0 of the byte array, we check the opCode value. Based in the value, we perform the following operations:-

Case 0: BOOL

When the OpCode is 0, the algorithm checks if the stack size is 1. If true, it sets `running` to `false`. Otherwise, it loops, restoring the state and updating `addr` if the byte at `addr` is 0 or 1.

Case 1: INT

For OpCode 1, the process is similar to Case 0. The stack size is checked, and if it is 1, `running` is set to `false`. If not, the algorithm loops, restoring and updating `addr` while the byte at `addr` is 0 or 1.

Case 64: ADD

- Check the values at `peekSecond()` and `peekThird()`.
- If both values are equal to 1:
  - Pop the top value from the stack.
  - Retrieve two integers from the stack using `getInt(pop()+ 1)`
  - Compute the addition of these two integers.
  - Allocate a new memory location for the result.
  - Store the result in the allocated memory.
  - Push the address of the allocated memory back onto the stack.
- If both of the initial conditions is not met:
  - Execute `backUp()` twice to revert the stack to its previous state.
- If either of the initial conditions is not met:
  - Execute `backUp()` once to revert the stack to its previous state.

Case 66: COND

- Check the value at `peekSecond()`.
- If the value is 0:
  - Pop the top value from the stack.
  - Retrieve `pop() + 1` value from the stack.
  - If the value is 1:
    * perform `backUp(), pop() and restore()` operations respectively
  - If the value is not 1:
    * Simply pop the top value from the stack.
- If the value at `peekSecond()` is not 0:
  - perform a `backUp()` operation.

Case 70: NOT

For OpCode 70, the algorithm simply performs a `backUp` operation.

Case 128: Application

For OpCode 128 (Application), first, it performs pop operation in the stack. Then, new values derived from the memory address at `addr + 5` and `addr + 1` are pushed into memory respectively.

Case 129: I

For OpCode 129, the algorithm simply performs a `pop()` operation.

Case 130: K

For OpCode 130, the algorithm performs sequantially `pop(), backUp(), pop(), restore()` operations.

Case 131: S

- Pop the top value from the stack, which is the combinator `S`.
- Retrieve the values of `x`, `y`, and `z`:
  - `x` is obtained from the current `stackTop`.
  - `y` is obtained from `peekSecond()`.
  - `z` is obtained from `peekThird()`.
- Allocate 9 bytes of memory to store the new address:
  - Store the opCode for the application (byte `128`) at the beginning of the allocated space.
  - Store `y` at offset 1 from the start of the allocated space.
  - Store `z` at offset 5 from the start of the allocated space.
- Perform the following operations:
  - Back up `x` to preserve its value.
  - Pop `y` from the stack.
  - Back up `z` to preserve its value.
  - Push the newly allocated address (containing the application of `y` and `z`) onto the stack.
  - perform `restore()` to restore 'z'
  - perform `restore()` to restore 'x'

Case 132: Y

- Pop the top value, i.e. 'Y' from the stack (the 'Y' combinator).

- Retrieve the value, i.e ‘x‘ of memory from the stackTop position.
- Allocate 9 bytes in memory and store:
  - The opCode ‘128‘ at the start.
  - The address ‘Y‘ at offset 1.
  - The value ‘x‘ at offset 5.
- Perform `backUP()` operation, push the new address onto stack and then perform `restore` operation.

Case 133: JMP

- Pop the top value, i.e. ‘JMP‘ from the stack (the ‘NTERM‘ combinator).
- push the new address onto stack.

please note, for the remaining opCodes i.e., 72 (SUB), 69 (MULT), 67 (EQ), 68 (LEQ), 65 (AND), 71 (OR), the steps are similar to the case 64 (ADD) as listed above except the respective operation for each combinator i.e for (69) MULT, steps are all the same except we perform multiplication of two integers.

Finally, two other methods are implemented for reading the result from the address stack and to identify whether the result is a constant or not. These methods are detailed below:

opCodeIsConst(byte b): This method determines whether the OpCode `b` represents a constant. OpCodes for constants are in the range $0-63$. To check this, the method evaluates `(b & 0b11000000) == 0`, which verifies that the two highest bits of the byte are 0.

readResult(): this method just invoke the opCodeIsConst methods. Depending on the output, it returns either an Integer or a boolean constant as the expected result.

4.3 Program Execution in x86 machine code

We have translated our program into assembler code by writing an assembler program, as shown in Appendix A.2. This assembler implementation mirrors the behavior of our Java implementation of the abstract machine (see Appendix A.1). In essence, the assembler template (see Appendix A.2) is functionally equivalent to the abstract machine in Java (see Appendix A.1), but written in assembler.

Figure 4.5 illustrates the text files and packages used during the assembler code implementation of our programs. The process can be divided into several key steps:

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td></tr>
<tr><td>128</td><td colspan="4">9</td><td colspan="4">118</td><td>128</td><td colspan="4">18</td><td colspan="4">103</td><td>128</td><td colspan="4">27</td><td colspan="4">28</td><td>131</td><td>128</td><td></td></tr>
<tr><td>30</td><td>31</td><td>32</td><td>33</td><td>34</td><td>35</td><td>36</td><td>37</td><td>38</td><td>39</td><td>40</td><td>41</td><td>42</td><td>43</td><td>44</td><td>45</td><td>46</td><td>47</td><td>48</td><td>49</td><td>50</td><td>51</td><td>52</td><td>53</td><td>54</td><td>55</td><td>56</td><td>57</td><td>58</td><td>59</td></tr>
<tr><td colspan="3">37</td><td colspan="4">58</td><td>128</td><td colspan="4">46</td><td colspan="4">47</td><td>131</td><td>128</td><td colspan="4">56</td><td colspan="4">57</td><td>130</td><td>64</td><td>128</td><td></td></tr>
<tr><td>60</td><td>61</td><td>62</td><td>63</td><td>64</td><td>65</td><td>66</td><td>67</td><td>68</td><td>69</td><td>70</td><td>71</td><td>72</td><td>73</td><td>74</td><td>75</td><td>76</td><td>77</td><td>78</td><td>79</td><td>80</td><td>81</td><td>82</td><td>83</td><td>84</td><td>85</td><td>86</td><td>87</td><td>88</td><td>89</td></tr>
<tr><td colspan="3">67</td><td colspan="4">102</td><td>128</td><td colspan="4">76</td><td colspan="4">77</td><td>131</td><td>128</td><td colspan="4">86</td><td colspan="4">87</td><td>130</td><td>128</td><td colspan="2"></td></tr>
<tr><td>90</td><td>91</td><td>92</td><td>93</td><td>94</td><td>95</td><td>96</td><td>97</td><td>98</td><td>99</td><td>100</td><td>101</td><td>102</td><td>103</td><td>104</td><td>105</td><td>106</td><td>107</td><td>108</td><td>109</td><td>110</td><td>111</td><td>112</td><td>113</td><td>114</td><td>115</td><td>116</td><td>117</td><td>118</td><td>119</td></tr>
<tr><td colspan="2">96</td><td colspan="4">97</td><td>69</td><td>1</td><td colspan="4">2</td><td>7</td><td>128</td><td colspan="4">112</td><td colspan="4">113</td><td>130</td><td>1</td><td colspan="4">3</td><td>1</td><td></td></tr>
<tr><td>120</td><td>121</td><td>122</td><td>123</td><td>124</td><td>125</td><td>126</td><td>127</td><td>128</td><td>129</td><td>130</td><td>131</td><td>132</td><td>133</td><td>134</td><td>135</td><td>136</td><td>137</td><td>138</td><td>139</td><td>140</td><td>141</td><td>142</td><td>143</td><td>144</td><td>145</td><td>146</td><td>147</td><td>148</td><td>149</td></tr>
<tr><td colspan="3">2</td><td>128</td><td colspan="4">103</td><td colspan="4">118</td><td>128</td><td colspan="4">58</td><td colspan="4">118</td><td>128</td><td colspan="4">102</td><td colspan="4">118</td></tr>
<tr><td colspan="3"></td><td colspan="9">Allocated in step 3</td><td colspan="9">Allocated in step 6</td><td colspan="9">Allocated in step 13</td></tr>
<tr><td>150</td><td>151</td><td>152</td><td>153</td><td>154</td><td>155</td><td>156</td><td>157</td><td>158</td><td>159</td><td colspan="20"></td></tr>
<tr><td>1</td><td colspan="4">4</td><td>1</td><td colspan="4">7</td><td colspan="20"></td></tr>
<tr><td colspan="5">Allocated in step 21</td><td colspan="5">Allocated in step 28</td><td colspan="20"></td></tr>
<tr><td colspan="5"></td><td colspan="5"></td><td colspan="20"></td></tr>
<tr><td colspan="5"></td><td colspan="5"></td><td colspan="20"></td></tr>
</table>

Figure 4.4: Final storage after execution for the program [main :: Int $=$ $(\lambda n.\ 2 \cdot n + 3)\ 2$]

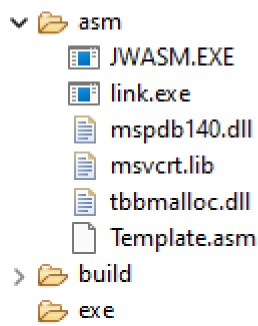

Figure 4.5: Necessary files and packages for the Assembler

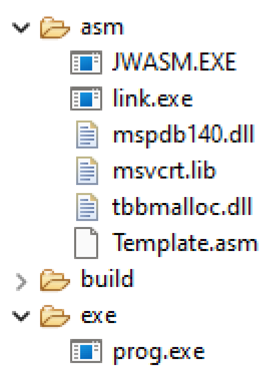

Figure 4.6: Executable version generation after the compilation

1. First, we created the assembler template (Section 4.3.1).
2. We translated the combinator term into an assembler program.
3. Specific content within the template (marked with hashtags) was replaced with concrete values.

4. The modified template was saved as a new file (i.e., $prog.asm$).

5. $JWasm.exe$ (Figure 4.5) took the $prog.asm$ file and produced a binary version, $prog.obj$, of the assembler code.

6. $Link.exe$ then created an executable ($prog.exe$)(see figure 4.6) from the binary file. The necessary libraries (files with $.dll$ and $.lib$ extensions, as shown in Figure 4.5) were required for the linker. External functions, such as program termination and $printf$, are defined within these libraries.

Further details on the assembler template and the generation of the assembler program are discussed in the following sections.

4.3.1 Assembler Template

As discussed above, this template mirrors the behavior of the Java implementation of the abstract machine. For better understanding, we can compare both the implementation in Appendix (see A.1 and A.2). Similar to Java, the template contains shorthand definitions for operations like $BOOL$, $INTC$, $ADDI$, etc. Following this, there are declarations of external functions that need to be called, such as $print$ and $exit$. These external functions are defined in the dll library (see Figure 4.5).

Additionally, the template defines several messages that might be printed during execution, such as "unexpected error," "overflow," "out of heap," etc. It also defines assembler variables like $stackCapacity$ and $heapCapacity$, with tags at the beginning and end (e.g., $\#stackCapacity\#$). These tags indicate that these variables will be replaced by concrete values, which is why it is referred to as a template. Another tag, $\#program\#$, will be replaced by the equivalent assembler program (see Figure 4.7) of the combinator.

Once the translation is complete, the linker ($Link.exe$) combines this with the necessary dll files to produce the final executable.

4.3.2 Assembler Program Generation

The assembler program is essentially an equivalent implementation of the memory structure of the abstract machine in Java, as illustrated in Figure 4.7. For instance, Figure 4.2 shows the memory of the abstract machine, which begins with the value 128, followed by a 4-byte address, 9. Similarly, as demonstrated in Figure 4.7, the first line of the assembler code starts with $prog\ byte\ 128$, and the second line reads $dword\ prog + 9$, where $dword$

denotes 4 bytes. This signifies that the next address is calculated from the label $prog$, offset by 9. The following instruction points to the address at $prog + 118$, and this pattern continues throughout the memory.

```
prog  byte 128
          dword prog+9
          dword prog+118
          byte 128
          dword prog+18
          dword prog+103
          byte 128
          dword prog+27
          dword prog+28
          byte 131
          byte 128
          dword prog+37
          dword prog+58
          byte 128
          dword prog+46
          dword prog+47
          byte 131
          byte 128
          dword prog+56
          dword prog+57
          byte 130
          byte 64
          byte 128
          dword prog+67
          dword prog+102
          byte 128
          dword prog+76
          dword prog+77
          byte 131
          byte 128
          dword prog+86
          dword prog+87
          byte 130
          byte 128
          dword prog+96
          dword prog+97
          byte 69
          byte 1
          dword 2
          byte 129
          byte 128
          dword prog+112
          dword prog+113
          byte 130
          byte 1
          dword 3
          byte 1
          dword 2
```

Figure 4.7: Assembler program for $(\lambda n.\ 2 \cdot n + 3)\ 2$

4.3.3 Compiling $[$main :: Int $= (\lambda n.\ 2 \cdot n + 3)\ 2]$

The process of generating the assembler version of a specific program involves several key steps. First, we create a template and modify it with concrete values and the assembler program (see figure 4.7) to form an updated version of the assembler template that incorporates our combinator term in assembler form. Using this updated template, along with an assembler, linker, and the necessary libraries (see figure 4.5), we then generate both the binary and executable versions of the program. The final executable file can be run externally to observe the output, as demonstrated in figure 4.8.

This process illustrates how any program can be translated into assembler code, and after processing the assembler file multiple times, we obtain an executable version that can be run from an external environment, such as the command prompt, to view the result. We refer to this entire procedure as ”compilation.”

As shown in figure 4.5, initially, there was no executable file in the ”exe” folder. However, after the compilation process, we can see that the executable file is generated and available in the same folder (see Figure 4.6).

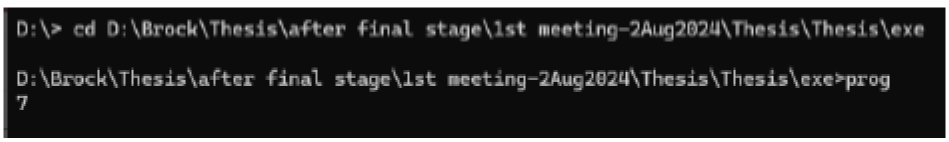

Figure 4.8: Result after running the executable version of$(\lambda n.\ 2 \cdot n + 3)\ 2$

4.4 Executing Program with Multiple Decalartions

To execute a program containing multiple decalartions (Equation 2.1) with its combinator form (Equation 3.1), we need to follow the steps outlined below:-

1. Program:

```
fact :: Int     Int = recf.  n . if n = 0 then 1 else n    f(
   n - 1);
succ :: Int     Int =  n . n + 1;
main :: Int = fact(succ 4);
```

2. Combinator Program:

```
fact = Y (S (K (S (S (S (K IF) (S (S (K EQUAL) I) (K 0))) (K
   1))))
          (S (K (S (S (K MULT) I)))
          (S (S (K S) (S (K K) I))
          (K (S (S (K SUB) I) (K 1))))))
```

```
succ = S (S (K ADD) I) (K 1)
main = fact (succ 4)
```

3. Storing:

```
[128, 0, 0, 0, 9, 0, 0, 0, 10, 132, 128,
0, 0, 0, 19, 0, 0, 0, 178, 128, 0, 0, 0, 28, 0, 0, 0, 29,
   131, 128, 0, 0, 0, 38,
0, 0, 0, 39, 130, 128, 0, 0, 0, 48, 0, 0, 0, 49, 131, 128,
0, 0, 0, 58, 0, 0, 0, 163, 128, 0, 0, 0, 67, 0, 0, 0, 68,
131, 128, 0, 0, 0, 77, 0, 0, 0, 98, 128, 0, 0, 0, 86, 0, 0,
0, 87, 131, 128, 0, 0, 0, 96, 0, 0, 0, 97, 130, 66, 128, 0,
0, 0, 107, 0, 0, 0, 148, 128, 0, 0, 0, 116, 0, 0, 0, 117,
131, 128, 0, 0, 0, 126, 0, 0, 0, 147, 128, 0, 0, 0, 135, 0,
0, 0, 136, 131, 128, 0, 0, 0, 145, 0, 0, 0, 146, 130, 67,
129, 128, 0, 0, 0, 157, 0, 0, 0, 158, 130, 1, 0, 0, 0, 0,
128, 0, 0, 0, 172, 0, 0, 0, 173, 130, 1, 0, 0, 0, 1, 128,
0, 0, 0, 187, 0, 0, 0, 248, 128, 0, 0, 0, 196, 0, 0, 0,
197, 131, 128, 0, 0, 0, 206, 0, 0, 0, 207, 130, 128, 0, 0,
0, 216, 0, 0, 0, 217, 131, 128, 0, 0, 0, 226, 0, 0, 0, 247,
128, 0, 0, 0, 235, 0, 0, 0, 236, 131, 128, 0, 0, 0, 245,
0, 0, 0, 246, 130, 69, 129, 128, 0, 0, 1, 1, 0, 0, 1, 72,
128, 0, 0, 1, 10, 0, 0, 1, 11, 131, 128, 0, 0, 1, 20, 0,
0, 1, 41, 128, 0, 0, 1, 29, 0, 0, 1, 30, 131, 128, 0, 0,
1, 39, 0, 0, 1, 40, 130, 131, 128, 0, 0, 1, 50, 0, 0, 1,
71, 128, 0, 0, 1, 59, 0, 0, 1, 60, 131, 128, 0, 0, 1, 69,
0, 0, 1, 70, 130, 130, 129, 128, 0, 0, 1, 81, 0, 0, 1, 82,
130, 128, 0, 0, 1, 91, 0, 0, 1, 132, 128, 0, 0, 1, 100,
0, 0, 1, 101, 131, 128, 0, 0, 1, 110, 0, 0, 1, 131, 128,
0, 0, 1, 119, 0, 0, 1, 120, 131, 128, 0, 0, 1, 129, 0,
0, 1, 130, 130, 72, 129, 128, 0, 0, 1, 141, 0, 0, 1, 142,
130, 1, 0, 0, 0, 1, 128, 0, 0, 1, 156, 0, 0, 1, 197,
128, 0, 0, 1, 165, 0, 0, 1, 166, 131, 128, 0, 0, 1, 175,
0, 0, 1, 196, 128, 0, 0, 1, 184, 0, 0, 1, 185, 131,
128, 0, 0, 1, 194, 0, 0, 1, 195, 130, 64, 129, 128,
0, 0, 1, 206, 0, 0, 1, 207, 130, 1, 0, 0, 0, 1,
128, 0, 0, 1, 221, 0, 0, 1, 226, 133, 0, 0, 0, 0,
128, 0, 0, 1, 235, 0, 0, 1, 240, 133, 0, 0, 1,
147, 1, 0, 0, 0, 4]
```

4. Assembler Program:

```
_fact   byte 128
                dword _fact+9
                dword _fact+10
                byte 132
                byte 128
                dword _fact+19
                dword _fact+178
                byte 128
                dword _fact+28
                dword _fact+29
                byte 131
                byte 128
                dword _fact+38
                dword _fact+39
                byte 130
                byte 128
                dword _fact+48
                dword _fact+49
                byte 131
                byte 128
                dword _fact+58
                dword _fact+163
                byte 128
                dword _fact+67
                dword _fact+68
                byte 131
                byte 128
                dword _fact+77
                dword _fact+98
                byte 128
                dword _fact+86
                dword _fact+87
                byte 131
                byte 128
                dword _fact+96
                dword _fact+97
                byte 130
```

```
                byte 66
                byte 128
                dword _fact+107
                dword _fact+148
                byte 128
                dword _fact+116
                dword _fact+117
                byte 131
                byte 128
                dword _fact+126
                dword _fact+147
                byte 128
                dword _fact+135
                dword _fact+136
                byte 131
                byte 128
                dword _fact+145
                dword _fact+146
                byte 130
                byte 67
                byte 129
                byte 128
                dword _fact+157
                dword _fact+158
                byte 130
                byte 1
                dword 0
                byte 128
                dword _fact+172
                dword _fact+173
                byte 130
                byte 1
                dword 1
                byte 128
                dword _fact+187
                dword _fact+248
                byte 128
                dword _fact+196
```

```
        dword _fact+197
        byte 131
        byte 128
        dword _fact+206
        dword _fact+207
        byte 130
        byte 128
        dword _fact+216
        dword _fact+217
        byte 131
        byte 128
        dword _fact+226
        dword _fact+247
        byte 128
        dword _fact+235
        dword _fact+236
        byte 131
        byte 128
        dword _fact+245
        dword _fact+246
        byte 130
        byte 69
        byte 129
        byte 128
        dword _fact+257
        dword _fact+328
        byte 128
        dword _fact+266
        dword _fact+267
        byte 131
        byte 128
        dword _fact+276
        dword _fact+297
        byte 128
        dword _fact+285
        dword _fact+286
        byte 131
        byte 128
```

```
            dword _fact+295
            dword _fact+296
            byte 130
            byte 131
            byte 128
            dword _fact+306
            dword _fact+327
            byte 128
            dword _fact+315
            dword _fact+316
            byte 131
            byte 128
            dword _fact+325
            dword _fact+326
            byte 130
            byte 130
            byte 129
            byte 128
            dword _fact+337
            dword _fact+338
            byte 130
            byte 128
            dword _fact+347
            dword _fact+388
            byte 128
            dword _fact+356
            dword _fact+357
            byte 131
            byte 128
            dword _fact+366
            dword _fact+387
            byte 128
            dword _fact+375
            dword _fact+376
            byte 131
            byte 128
            dword _fact+385
            dword _fact+386
```

```
                byte 130
                byte 72
                byte 129
                byte 128
                dword _fact+397
                dword _fact+398
                byte 130
                byte 1
                dword 1
_succ   byte 128
                dword _succ+9
                dword _succ+50
                byte 128
                dword _succ+18
                dword _succ+19
                byte 131
                byte 128
                dword _succ+28
                dword _succ+49
                byte 128
                dword _succ+37
                dword _succ+38
                byte 131
                byte 128
                dword _succ+47
                dword _succ+48
                byte 130
                byte 64
                byte 129
                byte 128
                dword _succ+59
                dword _succ+60
                byte 130
                byte 1
                dword 1
_main   byte 128
                dword _main+9
                dword _main+14
```

```
                byte 133
                dword _fact
                byte 128
                dword _main+23
                dword _main+28
                byte 133
                dword _succ
                byte 1
                dword 4
```

5. Update the template in Appendix A.2 with the assembler code from step 4 and other concrete values following the rules 4.3.3

6. Executing the exe file after compilation

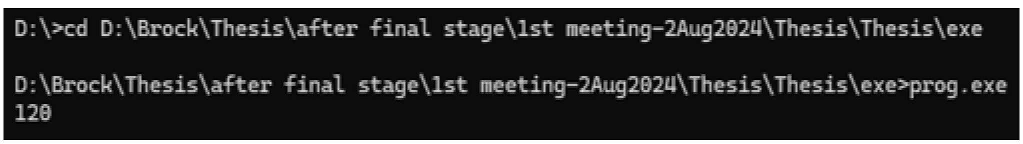

Figure 4.9: Result after running the executable version

Chapter 5

Implementation

For the compilation of PCF$^+$, we utilized the following technologies:

1. Eclipse (Version: 2023-03 (4.27.0))

2. JDK 21

3. Jarsec 3.1

Below is a brief discussion on the Jarsec package and the parser classes we created to implement the parsers.

5.1 Jparsec

Jarsec is a recursive-descent parser combinator framework for Java, modeled after Haskell's Parsec on the Java platform [9]. It features operator precedence grammar, accurate error location with customizable error messages, and a rich set of pre-defined reusable combinator functions. Its declarative API closely resembles Backus-Naur Form (BNF), facilitating clear and concise parser definitions.

In a typical parser program written in Jarsec, we need to create and combines various `Parser` object, each representing a piece of parsing logic. Jarsec provides key classes for constructing parsers according to the production rules (Section 5.1.2) of the grammar:

5.1.1 Key classes

- `Parser`: Encapsulates a piece of parsing logic. Simple `Parser` objects can be combined to create more complex parsers.

- `Parsers`: Implementations of common parsers.

- `Scanners`: Parses the source string and recognizes patterns.
- `Terminals`: `Terminals`: Provides tokenizers and lexers for common terminals, including identifiers, integers, and scientific numbers.
- `OperatorTable`: Supports operator precedence grammars by allowing the programmer to declare operators, which the framework uses to construct a full-blown expression parser.

Once a `Parser` object is created, it can be used as follows:

```
parser.parse("program to be parsed");
```

5.1.2 Production rules

Before discussing production rules, it is important to familiarize ourselves with the combinators used in the jparsec framework.

- `or`: Logical alternative combinator. The production rule $X ::= Y|Z$ can be represented as:

  ```
  Parser<className> x = Parsers.or(y,z);
  ```

- `sequence`: Sequential combinator. The production rule $X ::= YZ$ can be represented as:

  ```
  Parser<className> x = Parsers.sequence(y,z);
  ```

- `map/sequence`: These combinators allow performing computations or building objects based on recognized grammar. Using parser results of types Y and Z to create an object of type A:

  ```
  Parser<className> x = Parsers.sequence(y, z)
                  .map(new Map3<Y, Z>() {
                      public A map(Y y, Z z) {
                          return new X(y,z);
                      }
                  });
  ```

- `many/many1`: Implement the ”Kleene star” and ”Kleene cross” logic in BNF. The production rule $X ::= Y^*$ is represented as:

```
Parser<className> name = ...;
Parser<Void> x = name.skipMany();
```

  or

```
Parser<className> name = ...;
Parser<List<className>> x = name.many();
```

  where the latter will return a list of `className` objects as the parser result.

- `lazy`: Allows a parser to reference a parser that will be set later, useful for recursive production rules.

In jparsec, production rules define how complex language constructs are built from simpler elements. They simplify language structures into manageable components, guiding the parser in syntax recognition and interpretation. These rules are crucial during the syntactical analysis phase of a two-pass parser system. A two-pass parser processes input in two stages:

1. **Lexical Analysis**: Scans the source code to convert it into tokens, such as keywords and identifiers. A lexer or scanner handles this phase by removing unnecessary details like whitespace and comments and categorizing the text into meaningful symbols.

2. **Syntactical Analysis**: Takes the tokens from lexical analysis and organizes them into a syntax tree, reflecting the grammatical structure based on production rules. A parser manages this phase, ensuring that the token sequence adheres to the language's grammatical rules.

5.2 Parser classes

We have defined four types of parser classes: ‘TypeParser‘, ‘PCFTermParser‘,‘ProgramParser‘, and ‘CombinatorParser‘, each responsible for parsing types (Section 2.1.1), PCFTerms

(Section 2.1.2), programs (Section 2.1.3), and combinators (Section 3.1.1), respectively. When parsing input for both a ‘PCFTerm‘ and a ‘Type‘, we utilize the ‘OperatorTable‘ to declare operator precedences and associativities, thereby constructing parsers based on these declarations. Below, we provide snapshots of each parser to give a clear overview of their implementation. This will help in understanding how these parsers are constructed and utilized, building on the knowledge gained in the previous chapters.

5.2.1 TypeParser

We have constructed a parser object for types (Section 2.1.1). In this parser, we have utilized ‘OperatorTable‘ for the various operators used in defining types for the programs. For example, among the operators $\to$ and $*$, $*$ has the highest precedence as defined.

```
public Parser<Type> getParser( Terminals operators) {
    Parser.Reference<Type> ref = org.jparsec.Parser.newReference();
    Parser<Type> term =
            ref.lazy().between(operators.token("("), operators.token(")"))
            .or(operators.token("Int").retn(new IntType()))
            .or(operators.token("Bool").retn(new BoolType()))
            .or(Terminals.Identifier.PARSER.map(TypeVariable::new));
    Parser<Type> parser = new OperatorTable<Type>()
        .infixr(operators.token("->").retn((l,r) -> new FunctionType(l,r)), 3)
        .infixr(operators.token("*").retn((l,r) -> new ProductType(l,r)), 5)
        .build(term);
    ref.set(parser);
```

Figure 5.1: snapshot of the type’s parser

5.2.2 PCFTermParser

As we can see in Figure 5.2, to create the parser object for a PCFTerm in Java, we have utilized key classes, combinators, and production rules from the jparsec framework as described above (Section 5.1). For each of the integer and boolean operations, along with the comparisons, ‘OperatorTable‘ is employed to conveniently set operator precedence and associativities for the different operators used in our PCFTerm, such as $+, -, *, <$ $, \leq$, and, or, not.

5.2.3 ProgramParser

The `ProgramParser` utilizes both the typeParser and the PCFTermParser to parse a program (see Section 2.1.3) containing one or more declarations (see Section 2.1.3). Figure 5.3

```
public Parser<PCFTerm> getParser(Set decls, Terminals operators) {
    Parser.Reference<PCFTerm> ref = org.jparsec.Parser.newReference();
    Parser<PCFTerm> term =
            ref.lazy().between(operators.token("("), operators.token(")"))
            .or(Terminals.Identifier.PARSER.map(s -> { if (decls.contains(s))
                return new NamedTerm(s); else return new Variable(s); }))
            .or(operators.token("True").retn(new True()))
            .or(operators.token("False").retn(new False()))
            .or(Parsers.sequence(operators.token("if"),ref.lazy(),
            operators.token("then"),ref.lazy(),operators.token("else"),
            ref.lazy(),(t1,p1,t2,p2,t3,p3) -> new Conditional(p1,p2,p3)))
            .or(Terminals.IntegerLiteral.PARSER.map(s -> new IntLiteral(Integer.valueOf(s))))
            .or(Parsers.sequence(operators.token("\u03BB"),Terminals.Identifier.PARSER,
            operators.token("."),ref.lazy(),(s1,s2,s3,t) -> new Abstraction(s2,t)))
            .or(Parsers.sequence(operators.token("rec"),
            Terminals.Identifier.PARSER,operators.token("."),ref.lazy(),
            (s1,s2,s3,t) -> new Recursion(s2,t)));
    Parser<PCFTerm> typeTerm = term.many1().map(l -> makeApplications(l));
    Parser<PCFTerm> parser = new OperatorTable<PCFTerm>()
        .infixr(operators.token("or").retn((l,r) -> new Or(l,r)),Or.precedence)
        .infixr(operators.token("and").retn((l,r) -> new And(l,r)),And.precedence)
        .prefix(operators.token("not").retn((l) -> new Not(l)), Not.precedence )
        .infixr(operators.token("=").retn((l,r) -> new Equal(l,r)), Equal.precedence)
        .infixr(operators.token("<=").retn((l,r) -> new LEqual(l,r)), LEqual.precedence)
        .infixr(operators.token("+").retn((l,r) -> new Addition(l,r)), Addition.precedence)
        .infixn(operators.token("-").retn((l,r) -> new Subtraction(l,r)), Subtraction.precedence)
        .infixr(operators.token("*").retn((l,r) -> new Mult(l,r)), Mult.precedence)
        .build(typeTerm);
    ref.set(parser);
    return parser;
}
```

Figure 5.2: Snapshot of the PCFTerm's parser

below provides a clear view of how a program with multiple declarations is parsed with the help of these two parsers (see Sections 5.2.1 and 5.2.2).

```
public Parser<Program> getParser() {
    BiFunction<BiFunction,Program,Parser<Program>> frec = (f,p) ->
        Parsers.sequence(Terminals.Identifier.PARSER,
        OPERATORS.token("::"),
        TypeParser.getTypeParser().getParser(OPERATORS),
        OPERATORS.token("="),
        PCFTermParser.getPCFTermParser().getParser(p.nameSet(),OPERATORS),
        OPERATORS.token(";"),
        (n,s,t,s1,body,s2) -> {
            Declaration decl = new Declaration(n,t,body);
            p.addDeclaration(decl.getName(),decl);
            return p;
        }).next(q -> (Parser<Program>) f.apply(f,q))
        .or(Parsers.constant(p));
    return frec.apply(frec,new Program());
}
```

Figure 5.3: Snapshot of the Program's parser

5.2.4 CombinatorParser

This parser is relatively straightforward. We simply need to follow the key classes and combinators defined in the jparsec package (Section 5.1) to build the parser for the combinator terms (Section 3.1.1).

```
public Parser<CombinatorTerm> getParser(Set decls, Terminals operators) {
    Parser.Reference<CombinatorTerm> ref = org.jparsec.Parser.newReference();
    Parser<CombinatorTerm> term =
            ref.lazy().between(operatorsC.token("("), operatorsC.token(")"))
            .or(Terminals.Identifier.PARSER.map(s -> { if (decls.contains(s))
            return new Combinators.NamedTerm(s); else return new VARIABLE(s); }))
            .or(operatorsC.token("TRUE").retn(new TrueTerm()))
            .or(operatorsC.token("FALSE").retn(new FalseTerm()))
            .or(operatorsC.token("S").retn(new STerm()))
            .or(operatorsC.token("K").retn(new KTerm()))
            .or(operatorsC.token("I").retn(new ITerm()))
            .or(operatorsC.token("Y").retn(new YTerm()))
            .or(operatorsC.token("ADD").retn(new AddTerm()))
            .or(operatorsC.token("MULT").retn(new MultTerm()))
            .or(operatorsC.token("SUB").retn(new SubTerm()))
            .or(operatorsC.token("AND").retn(new AndTerm()))
            .or(operatorsC.token("OR").retn(new OrTerm()))
            .or(operatorsC.token("NOT").retn(new NotTerm()))
            .or(operatorsC.token("EQUAL").retn(new EqualTerm()))
            .or(operatorsC.token("LEQUAL").retn(new LEqualTerm()))
            .or(operatorsC.token("IF").retn(new ConditionalTerm()))
            .or(Terminals.IntegerLiteral.PARSER.map(s -> new
            INTTerm(Integer.valueOf(s))));
             Parser<CombinatorTerm> combTerm =
            term.many1().map(l -> makeApplications(l));
            ref.set(combTerm);
            return combTerm;
}
```

Figure 5.4: Snapshot of the combinator's parser

5.3 User Interface

The user interface is simple and straightforward, implemented using Swing. It includes an input area for entering the program and an output area to display the corresponding output. Users have three options for a given program: `Check`, `Compile`, and `Run`. Each of these options will be discussed in detail below.

Check

This option checks if a program is syntactically correct and free of any type errors. For example, Figure 5.5 shows a correctly formatted input program, whereas Figure 5.6 displays an incorrect program with the error reason indicated in the output message area.

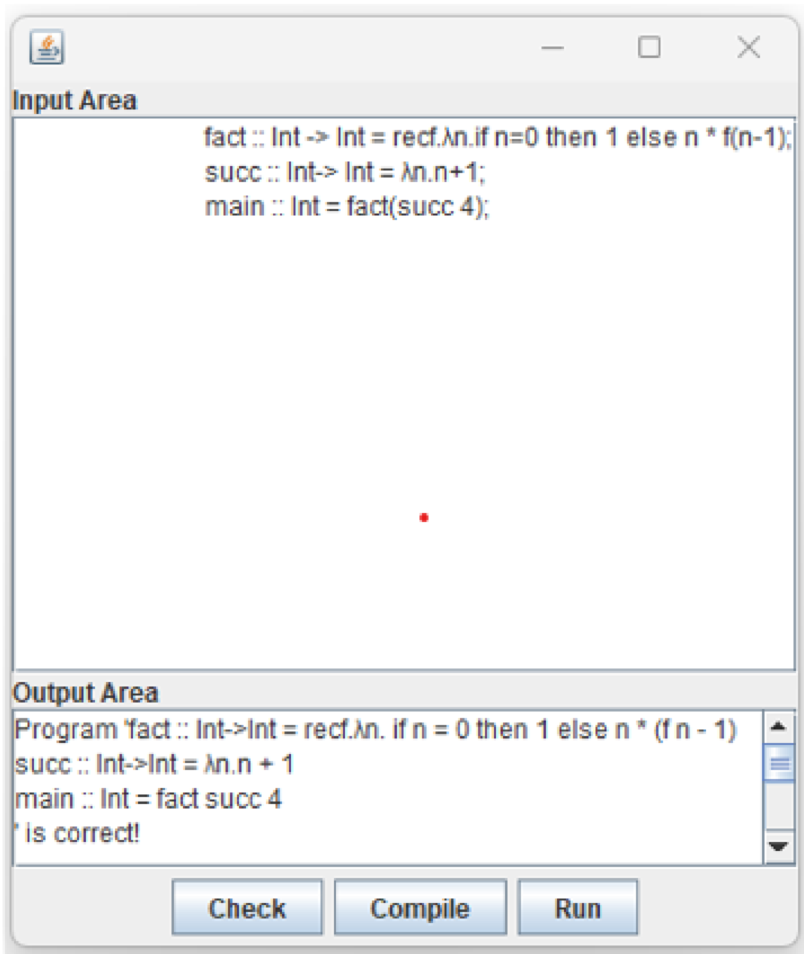

Figure 5.5: User Interface: Check the program

Compile

If the entered program is correct (as verified using the Section 5.3 option described above), it can then be successfully compiled into an assembler version using this option, as shown in Figure 5.7. After compilation, the program's output can be examined by following the steps outlined in Section 4.3.3.

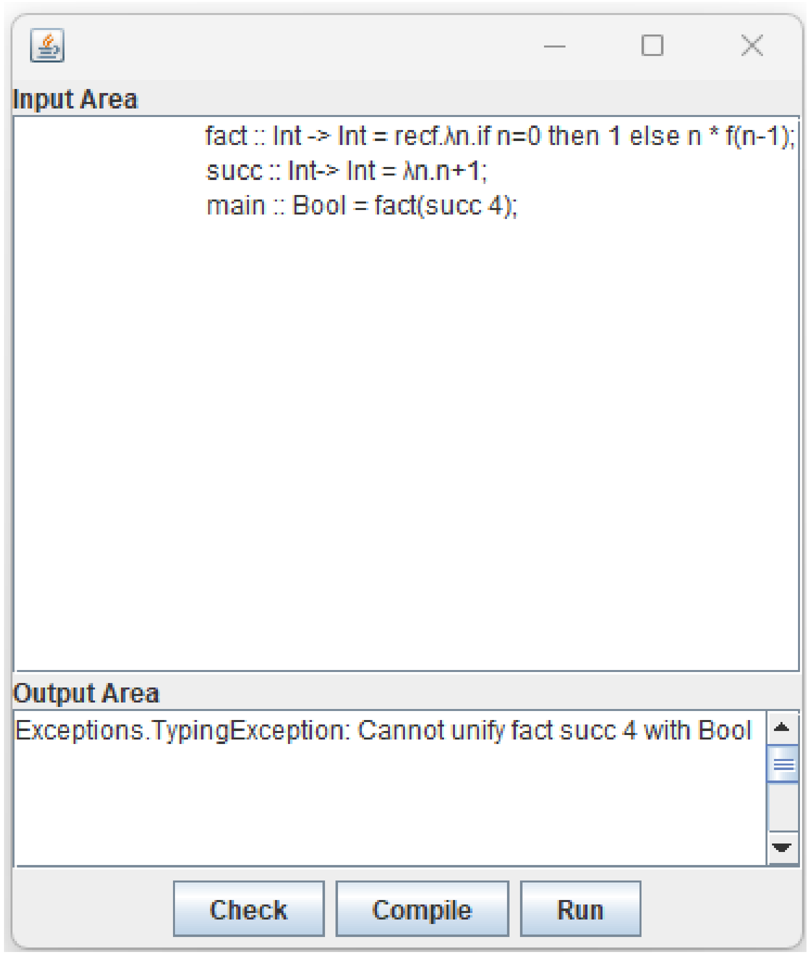

Figure 5.6: User Interface: Check the program (Error)

Run

The user can view the results of the entered program executed in Java using this option, as shown in Figure 5.8.

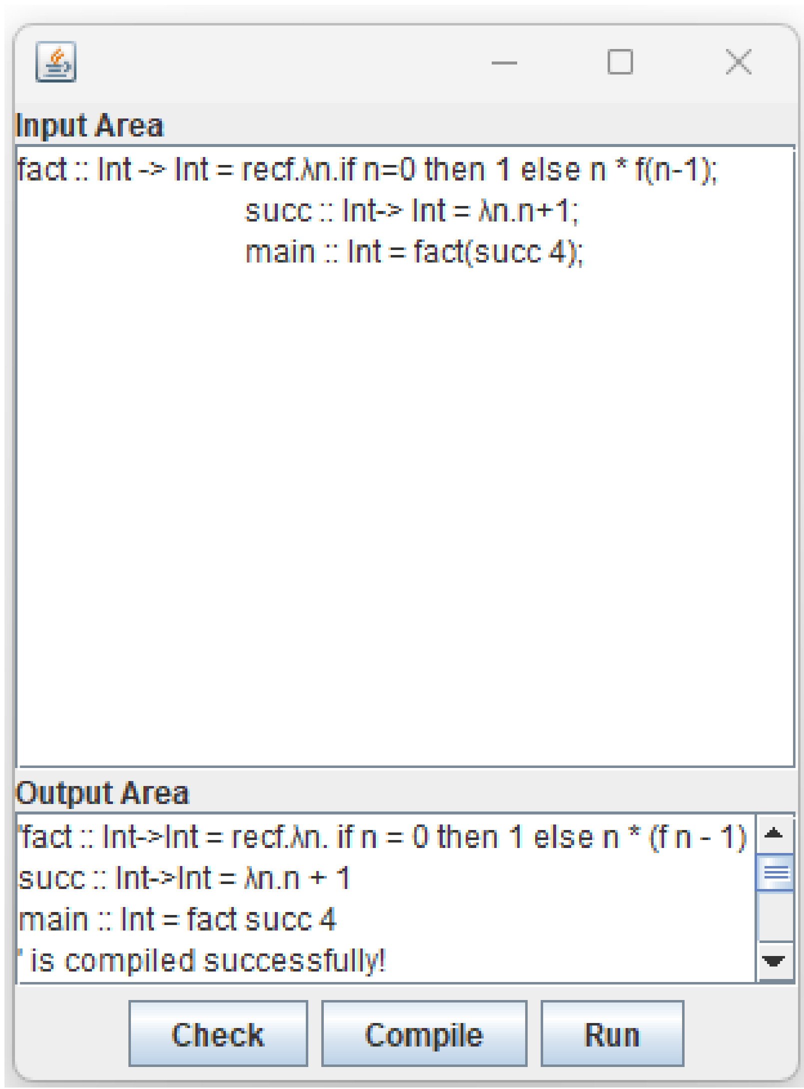

Figure 5.7: User Interface: compile the program

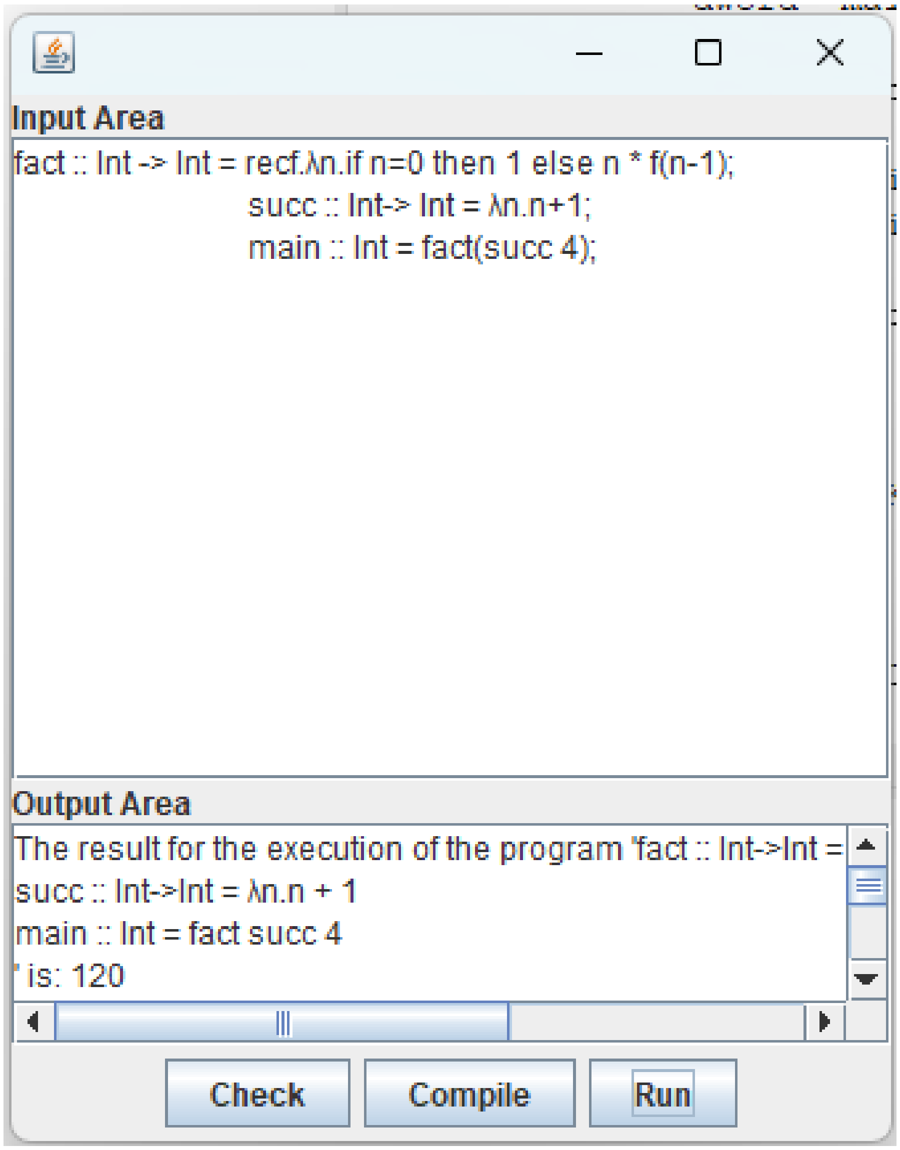

Figure 5.8: User Interface: run the program

Chapter 6

Conclusion and Future Work

In our thesis, we accomplished our goals through a series of structured steps. First, we chose Java as our development environment and selected the existing functional language, PCF, to enrich it by adding more types or modifying the existing ones. We defined several Java parsers (see Chapter 5) to conveniently construct programs and types conveniently, ensuring strict type compatibility between operands and leveraging mathematical functions to create reliable and maintainable code. This restricted operations to operands of compatible types.

Next, within the same Java environment, we translated this language into an appropriate combinator language. This involved mapping the program constructs into combinators to simplify the programs and facilitate easier manipulation and execution. We then designed an abstract machine based on combinators to execute these programs, managing memory, instructions, and data in a combinator-based format. This machine would be implemented in software as a virtual machine and could potentially be built as a custom processor, integrating seamlessly with functional programming principles. Subsequently, we would translate the abstract machine into assembly language, creating an executable version of the initial program that can be run directly on x86 hardware or in an emulator.

Our future work includes incorporating custom data types similar to Haskell's rich type system, necessitating effective handling of constructors to support user-defined types and enabling more expressive and type-safe code. Additionally, expanding the combinator set by integrating additional combinators, such as B, C, K, and W, will enrich the language's expressiveness through a smarter translation mechanism. This mechanism will leverage these combinators to generate more efficient programs, optimizing performance and resource usage. These enhancements aim to improve the language's capabilities and efficiency, providing users with more powerful tools for functional programming.

Bibliography

[1] Erlang Community. Erlang programming language. https://www.erlang.org/, 2024. Accessed: December 24, 2024.

[2] Haskell Community. Haskell language. https://www.haskell.org/, 2024. Accessed: December 24, 2024.

[3] Wikipedia contributors. Functional programming. https://en.wikipedia.org/wiki/Functional_programming, 2024. Accessed: December 24, 2024.

[4] Wikipedia Contributors. Lisp programming language. https://en.wikipedia.org/wiki/Lisp_(programming_language), 2024. Accessed: December 24, 2024.

[5] Wikipedia contributors. Von neumann architecture. https://en.wikipedia.org/wiki/Von_Neumann_architecture, 2024. Accessed: December 24, 2024.

[6] Jacek Gibert. Functional programming with combinators. *Journal of Symbolic Computation*, 4(3):269–293, 1987.

[7] Paul Hudak and David Kranz. A combinator-based compiler for a functional language. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, page 122–132, New York, NY, USA, 1984. Association for Computing Machinery.

[8] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, page 1–10, New York, NY, USA, 1982. Association for Computing Machinery.

[9] JParsec. Github. Retrieved November 08, 2024, from https://github.com/jparsec/jparsec.

[10] Ben Lynn. A combinatory compiler. Retrieved November 08, 2024 from https://crypto.stanford.edu/ blynn/lambda/sk.html.

[11] Alberto Martelli and Ugo Montanari. Unification in linear time and space: a structured presentation. 1976.

[12] Peter M. Maurer and Arthur E. Oldehoeft. The use of combinators in translating a purely functional language to low-level data-flow graphs. *Computer Languages*, 8(1):27–45, 1983.

[13] Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford, CA, USA, 1972.

[14] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., USA, 1987.

[15] G.D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[16] J. A. Robinson. Computational logic: the unification computation.

[17] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

Appendix A

Additional Experimental Analysis

A.1 Implementation In Java

Listing A.1: AbstractMachine.java

```java
package Webasm;

import Combinators.CombinatorProgram;
import Combinators.CombinatorTerm;
import java.util.HashMap;
import java.util.Map;

public class AbstractMachine {

    public static final byte BOOL   = (byte) 0;
    public static final byte INT    = (byte) 1;

    public static final byte ADD    = (byte) 64;
    public static final byte AND    = (byte) 65;
    public static final byte COND   = (byte) 66;
    public static final byte EQ     = (byte) 67;
    public static final byte LEQ    = (byte) 68;
    public static final byte MUL    = (byte) 69;
    public static final byte NOT    = (byte) 70;
    public static final byte OR     = (byte) 71;
    public static final byte SUB    = (byte) 72;

```

```
    public static final byte APP    = (byte) 128;
    public static final byte I      = (byte) 129;
    public static final byte K      = (byte) 130;
    public static final byte S      = (byte) 131;
    public static final byte Y      = (byte) 132;
    public static final byte JMP    = (byte) 133;

    private int[] backupStack;
    private final int stackCapacity;

    private int stackPointer;
    private int backupPointer;

    private int startAddress;

    private final Memory memory;

    private boolean flagRUN;

    public AbstractMachine(int stackCapacity, int heapCapacity) {
        this.stackCapacity = stackCapacity;
        this.memory = new Memory();
    }

    public void store(CombinatorTerm t) {
        store(t,new HashMap());
        startAddress = 0;
    }

    private void store(CombinatorTerm t, Map<String,Integer>
       labels) {
        t.storeInMem(memory,labels);
    }

    public void store(CombinatorProgram prog) {
        Map<String,Integer> labels = new HashMap();
        for(String name : prog.getNames()) {
            labels.put(name,memory.allocate(0));
```

```
            if (name.equals("main")) startAddress = labels.get("main");
            prog.getTerm(name).storeInMem(memory,labels);
        }
        System.out.println("byte arrary: "+memory.toString());
    }

    public void execute() {
        allocateStack();
        push(startAddress);
        flagRUN = true;
        while (flagRUN) {
            int addr = peek();
            byte opCode = memory.getByte(addr);
            switch (opCode) {
                case INT, BOOL -> {
                    if (stackSize()==1) {
                        flagRUN = false;
                    } else {
                        restore();
                    }
                }
                case ADD, MUL, SUB -> {
                    if (opCodeIsConst(memory.getByte(peekSecond()))) {
                        if(opCodeIsConst(memory.getByte(peekThird()))) {
                            remove();
                            int x = memory.getInt(pop()+1);
                            int y = memory.getInt(pop()+1);
                            int newAddress = memory.allocate(5);
                            memory.storeByte(newAddress,INT);
                            switch (opCode) {
                                case ADD -> {
                                    memory.storeInt(newAddress+1,x+y);
                                }
                                case MUL -> {
```

```
                                        memory.storeInt(newAddress+1,
                                            x*y);
                                    }
                                    case SUB -> {
                                        memory.storeInt(newAddress+1,
                                            x-y);
                                    }
                                }
                                push(newAddress);
                            } else {
                                backUp();
                                backUp();
                            }
                        } else {
                            backUp();
                        }
                    }
                    case AND, OR -> {
                        if (opCodeIsConst(memory.getByte(peekSecond()
                            ))) {
                            if(opCodeIsConst(memory.getByte(peekThird
                                ()))) {
                                remove();
                                byte x = memory.getByte(pop()+1);
                                byte y = memory.getByte(pop()+1);
                                int newAddress = memory.allocate(2);
                                memory.storeByte(newAddress,BOOL);
                                switch (opCode) {
                                    case AND -> {
                                        memory.storeByte(newAddress
                                            +1,(byte)(x&y));
                                    }
                                    case OR -> {
                                        memory.storeByte(newAddress
                                            +1,(byte)(x|y));
                                    }
                                }
                                push(newAddress);
```

```
                } else {
                    backUp();
                    backUp();
                }
            } else {
                backUp();
            }
        }
        case COND -> {
            if (opCodeIsConst(memory.getByte(peekSecond()))) {
                remove();
                byte cond = memory.getByte(pop()+1);
                if (cond==0) {
                    remove();
                } else {
                    int x = pop();
                    remove();
                    push(x);
                }
            } else {
                backUp();
            }
        }
        case EQ, LEQ -> {
            if (opCodeIsConst(memory.getByte(peekSecond()))) {
                if(opCodeIsConst(memory.getByte(peekThird()))) {
                    remove();
                    int x = memory.getInt(pop()+1);
                    int y = memory.getInt(pop()+1);
                    int newAddress = memory.allocate(2);
                    memory.storeByte(newAddress,BOOL);
                    switch (opCode) {
                        case EQ -> {
                            memory.storeByte(newAddress+1,(byte)(x==y? 1 : 0));
```

```java
                                }
                                case LEQ -> {
                                    memory.storeByte(newAddress
                                       +1,(byte)(x<=y? 1 : 0));
                                }
                            }
                            push(newAddress);
                        } else {
                            backUp();
                            backUp();
                        }
                    } else {
                        backUp();
                    }
                }
                case NOT -> {
                    if (opCodeIsConst(memory.getByte(peekSecond()
                       ))) {
                        remove();
                        byte x = memory.getByte(pop()+1);
                        int newAddress = memory.allocate(2);
                        memory.storeByte(newAddress,BOOL);
                        memory.storeByte(newAddress+1,(byte)((x
                           ==1?0:1)));
                        push(newAddress);
                    } else {
                        backUp();
                    }
                }
                case APP -> {
                    remove();
                    push(memory.getInt(addr+5));
                    push(memory.getInt(addr+1));
                }
                case I -> {
                    remove();
                }
                case K -> {
```

```
                    remove();
                    int x = pop();
                    remove();
                    push(x);
                }
                case S -> {
                    remove();
                    int x = pop();
                    int y = pop();
                    int z = pop();
                    int newAddress = memory.allocate(9);
                    memory.storeByte(newAddress,APP);
                    memory.storeInt(newAddress+1,y);
                    memory.storeInt(newAddress+5,z);
                    push(newAddress);
                    push(z);
                    push(x);
                }
                case Y -> {
                    remove();
                    int g = pop();
                    int newAddress = memory.allocate(9);
                    memory.storeByte(newAddress,APP);
                    memory.storeInt(newAddress+1,addr);
                    memory.storeInt(newAddress+5,g);
                    push(newAddress);
                    push(g);
                }
                case JMP -> {
                    remove();
                    push(memory.getInt(addr+1));
                }
                default -> throw new
                  UnsupportedOperationException("Unknown
                  operation code in abstract machine.");
            }
        }
    }
```

```

    public String readResult() {
        String result;
        int addr = pop();
        switch (memory.getByte(addr)) {
            case BOOL -> {
                result = memory.getByte(addr+1)==1? "True" : "
                    False";
            }
            case INT -> {
                result = Integer.toString(memory.getInt(addr+1));
            }
            default -> {
                result = "Unexpected error";
            }
        }
        return result;
    }

    private boolean opCodeIsConst(byte b) {
        return (b & 0b11000000) == 0;
    }

    private void allocateStack() {
        backupStack = new int[stackCapacity];
        stackPointer = 0;
        backupPointer = stackCapacity-1;
    }

    private void push(int value) {
        if (backupPointer < stackPointer) throw new
            StackOverflowError();
        backupStack[stackPointer] = value;
        stackPointer++;
    }

    private int pop() {
        stackPointer--;
```

```
        return backupStack[stackPointer];
    }

    private void remove() {
        stackPointer--;
    }

    private int peek() {
        return backupStack[stackPointer-1];
    }

    private int peekSecond() {
        return backupStack[stackPointer-2];
    }

    private int peekThird() {
        return backupStack[stackPointer-3];
    }

    private void backUp() {
        stackPointer--;
        int x = backupStack[stackPointer];
        backupStack[backupPointer] = x;
        backupPointer--;
    }

    private void restore() {
        backupPointer++;
        int x = backupStack[backupPointer];
        backupStack[stackPointer] = x;
        stackPointer++;
    }

    private int stackSize() {
        return stackPointer+stackCapacity-(backupPointer+1);
    }
}
```

sec

A.2 Implementation in Assembler Code

Listing A.2: Assembler Template

```
;--- Win32 Template for combinator compiler.
;--- assemble: jwasm -coff Test.asm
;--- link:     link Test.obj msvcrt.lib

    .386
    .MODEL flat, c
    option casemap:none

BOOL    equ 0
INTC    equ 1

ADDI    equ 64
ANDI    equ 65
COND    equ 66
EQI     equ 67
LEQ     equ 68
MULI    equ 69
NOTI    equ 70
ORI     equ 71
SUBI    equ 72

APP     equ 128
I       equ 129
K       equ 130
S       equ 131
Y       equ 132
MJMP    equ 133

printf  proto c :ptr byte, :vararg
malloc  proto c :dword
exit    proto c :dword

    .CONST

; messages
```

```

intMessage              db "%d",13,10,0
trueMessage             db "True",13,10,0
falseMessage            db "False",13,10,0
errorMessage            db "Unexpected error.",13,10,0
stackOverFlowMessage    db "Stack overflow.",0
outOfHeapMessage        db "Out of heap.",0

; important constant

stackCapacity   dword #stackCapacity#                ; must be dividable by 4
heapCapacity    dword #heapCapacity#

#program#

    .DATA

stackBaseAddr   dword ?
stackPointer    dword ?
backupPointer   dword ?
heapBaseAddr    dword ?
heapPointer     dword ?
flagRUN         byte 1

    .CODE

_allocateStack proc c
    push eax
    mov eax, stackCapacity
    invoke malloc, eax
    mov stackBaseAddr, eax
    mov stackPointer, eax
    add eax, stackCapacity                  ; eax set to highest dword address
    sub eax, 4                              ; in stack memory
    mov backupPointer, eax
    pop eax
```

```
    ret
_allocateStack endp

_allocateHeap proc c
    push eax
    mov eax, heapCapacity
    invoke malloc, eax
    mov heapBaseAddr, eax
    mov heapPointer, eax
    pop eax
    ret
_allocateHeap endp

_newOnHeap proc c
    push ebx
    mov ebx, heapPointer
    add ebx, eax
    sub ebx, heapBaseAddr
    .if (heapCapacity < ebx)
        invoke printf, addr outOfHeapMessage
        mov eax, 1
        invoke exit, eax
    .endif
    mov ebx, heapPointer
    add heapPointer, eax
    mov eax, ebx
    pop ebx
    ret
_newOnHeap endp

_push proc c
    push ebx
    mov ebx, stackPointer
    .if (backupPointer < ebx)
        invoke printf, addr stackOverFlowMessage
        mov eax, 1
        invoke exit, eax
    .endif
```

```
    mov [ebx], eax
    add stackPointer, 4
    pop ebx
    ret
_push endp

_pop proc c
    push ebx
    sub stackPointer, 4
    mov ebx, stackPointer
    mov eax, [ebx]
    pop ebx
    ret
_pop endp

_remove proc c
    sub stackPointer, 4
    ret
_remove endp

_peek proc c
    push ebx
    mov ebx, stackPointer
    sub ebx, 4
    mov eax, [ebx]
    pop ebx
    ret
_peek endp

_peekSecond proc c
    push ebx
    mov ebx, stackPointer
    sub ebx, 8
    mov eax, [ebx]
    pop ebx
    ret
_peekSecond endp
```

```
_peekThird proc c
    push ebx
    mov ebx, stackPointer
    sub ebx, 12
    mov eax, [ebx]
    pop ebx
    ret
_peekThird endp

_backup proc c
    push eax
    push ebx
    sub stackPointer, 4
    mov ebx, stackPointer
    mov eax, [ebx]
    mov ebx, backupPointer
    mov [ebx], eax
    sub backupPointer, 4
    pop ebx
    pop eax
    ret
_backup endp

_restore proc c
    push eax
    push ebx
    add backupPointer, 4
    mov ebx, backupPointer
    mov eax, [ebx]
    mov ebx, stackPointer
    mov [ebx], eax
    add stackPointer, 4
    pop ebx
    pop eax
    ret
_restore endp

_stackSize proc c
```

```
    mov eax, stackPointer
    add eax, stackCapacity
    sub eax, backupPointer
    shr eax, 2                  ; divide by 4
    dec eax
    ret
_stackSize endp

main proc c
    call _allocateStack
    call _allocateHeap
    mov eax, offset _main
    call _push
    .while(flagRUN != 0)
        call _peek
        mov ebx, eax
        mov al, [ebx]
        .if (al == BOOL  || al == INTC)
            call _stackSize
            .if (eax == 1)
                mov flagRUN, 0
            .else
                call _restore
            .endif
        .else
        .if (al == ADDI || al == MULI || al == SUBI)
            mov dl, al
            call _peekSecond
            mov al, [eax]
            and al, 11000000b
            .if (al == 0)
                call _peekThird
                mov al, [eax]
                and al, 11000000b
                .if (al == 0)
                    call _remove
                    call _pop
                    mov ebx, [eax+1]
```

```
                    call _pop
                    mov ecx, [eax+1]
                    mov eax, ebx
                    mov ebx, ecx
                    .if (dl == ADDI)
                        add eax, ebx
                    .else
                    .if (dl == MULI)
                        mul ebx
                    .else
                        sub eax, ebx
                    .endif
                    .endif
                    mov ebx, eax
                    mov eax, 5
                    call _newOnHeap
                    mov [eax], dword ptr INTC
                    mov [eax+1], ebx
                    call _push
                 .else
                    call _backup
                    call _backup
                .endif
            .else
                call _backup
            .endif
        .else
        .if (al == ANDI || al == ORI)
            mov dl, al
            call _peekSecond
            mov al, [eax]
            and al, 11000000b
            .if (al == 0)
                call _peekThird
                mov al, [eax]
                and al, 11000000b
                .if (al == 0)
                    call _remove
```

```
            call _pop
            mov bl, [eax+1]
            call _pop
            mov cl, [eax+1]
            mov al, bl
            mov bl, cl
            .if (dl == ANDI)
                and al, bl
            .else
                or al, bl
            .endif
            mov bl, al
            mov eax, 2
            call _newOnHeap
            mov [eax], dword ptr BOOL
            mov [eax+1], bl
            call _push
         .else
            call _backup
            call _backup
        .endif
    .else
        call _backup
    .endif
.else
.if (al == COND)
    mov dl, al
    call _peekSecond
    mov al, [eax]
    and al, 11000000b
    .if (al == 0)
        call _remove
        call _pop
        mov bl, [eax+1]
        .if (bl == 0)
            call _remove
        .else
            call _pop
```

```
                    call _remove
                    call _push
                .endif
            .else
                call _backup
            .endif
        .else
        .if (al == EQI)
            call _peekSecond
            call _pop
            mov ebx, [eax]
            call _peekSecond
            call _pop
            mov eax, [eax]
            .if (ebx == eax)
                call _push
                mov eax, 1
            .else
                call _push
                mov eax, 0
            .endif
        .else
        .if (al == LEQ)
            call _peekSecond
            call _pop
            mov ebx, [eax]
            call _peekSecond
            call _pop
            mov eax, [eax]
            .if (ebx <= eax)
                call _push
                mov eax, 1
            .else
                call _push
                mov eax, 0
            .endif
        .else
        .if (al == MJMP)
```

```
                call _pop
                mov ebx, [eax]
                jmp ebx
            .else
                call _backup
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
            .endif
        .endw
        mov eax, 0
        invoke exit, eax
main endp

_main proc c
        ret
_main endp

end main
```