A Compiler for a Functional Programming Language based on System F

Bachelor's Thesis DATX11-VT23-66

William Bodin
Martin Fredin
Samuel Hammersberg
Valter Miari
Victor Olin
Sebastian Selander

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023
www.chalmers.se

Bachelor's Thesis 2023

A Compiler for a Functional Programming Language based on System F

William Bodin
Martin Fredin
Samuel Hammersberg
Valter Miari
Victor Olin
Sebastian Selander

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

A Compiler for a Functional Programming Language based on System F

Supervisor: Patrik Jansson, Department of Computer Science and Engineering
Graded by teacher: Alex Gerdes, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Bachelor's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Acknowledgements

We would like to thank our supervisor Patrik Jansson for continuously providing us with inspiration, knowledge and guidance throughout the whole project.

Abstract

This thesis presents the design and implementation of a compiler for a functional programming language based on System F. The language, Churf, is designed solely for this project and the compiler implementation. Churf's compiler has a frontend, and a backend. The frontend of the compiler uses a lexer and parser generated by the BNF Converter (BNFC), which in turn provides an abstract syntax tree for the compiler's type checker. The type checker then produces an annotated abstract syntax tree for the compiler backend, which lifts lambda functions, transforms polymorphic functions into monomorphic functions, and generates LLVM IR code. The LLVM IR code is compiled and linked with the runtime library, producing an executable. The project successfully implements a compiler for a functional language, able to compile lambda functions, pattern matching, and data types. The data types are heap-allocated and garbage collected by the runtime library.

Sammandrag

Denna tes presenterar designen och implementationen av en kompilator för ett funktionellt programmeringsspråk baserat på System F. Språket Churf är designat endast för detta projekt och kompilatorn. Churfs kompilator har en framdel och en bakdel. Framdelen av kompilatorn är genererad av BNF Converter (BNFC), och den utför lexikalisk och syntaktisk analys. Resultat är ett abstrakt syntaxträd som sedan annoteras med typer via typcheckaren. Därefter tar backdelen av kompilatorn över; lambdafunktioner lyfts bort, polymorfiska funktioner omvandlas till monomorfiska funktioner, och LLVM IR-koden generaras. Slutligen länkas den genererade koden med körtidsbiblioteket, vilket skapar en körbar fil. Projektet genomför framgångsrikt en kompilator för ett funktionellt språk, som kan kompilera lambdafunktioner, mönstermatchning, och datatyper. Datatyperna är heapallokerade och skräpsamlas av körtidsbiblioteket.

List of Figures

Contents

1 Introduction

The development of programming languages is intimately related to compiler development. As programming languages evolve to introduce abstractions, algorithms need to be devised and implemented. To properly understand the inner workings of programming languages, it is important to explore the field of compiler technology. This thesis presents the design and implementation of a compiler and a runtime library for a functional programming language.

1.1 Background

In order to execute a computer program, the source code needs to be translated into machine code that the computer processor can understand. This translation is usually the purpose of compilers. Compilers have been constructed since the mid-50s and early compilers were often designed as monolithic systems. They received source code in a specific language as input and compiles it to machine code for a specific processor architecture (Cooper & Torczon, 2003). Modern compilers, however, are designed to be more flexible and split the goal of translation into smaller steps, while simultaneously supporting more than one architecture.

Each step represents a pass over the source code, optimizing or translating it to some intermediate representation for the next step of the compiler. In modern compiler technology, there exists an idea of a trichotomy of steps. The compilation is split into three passes; frontend, middle, and backend (Cooper & Torczon, 2003). These passes perform syntactic and semantic analysis, intermediate code generation and optimization, and code generation for the target architecture.

It is important to note that the level of difficulty in implementing a compiler varies depending on the abstraction distance the compiler translates between (Ranta, 2012). Higher-level languages use abstractions to make the language more productive and safer. However, these abstractions are not available in lower-level languages and therefore have to be compiled into an equivalent representation. The implementations of abstractions need to be taken into consideration when constructing a programming language and a compiler for it.

1.2 Purpose

The goal of this project is to design and implement a compiler for a strictly evaluated functional programming language where the compilation target is LLVM intermediate representation (LLVM IR). Language design is not the focus of the project and the resulting language, hereafter called Churf, may be viewed as a side effect of the compiler design. Although, it is worth mentioning that some design choices of Churf impacted the implementation of the compiler. The source language is based on the polymorphic lambda calculus System F.

The project focuses on three key topics of a compiler; semantic analysis, intermediate representation, and code generation. In addition to the compiler, a separate goal of the project is to create a runtime library for the language, which the compiler incorporates in the generated LLVM IR code to support heap allocation and garbage collection.

1.3 Delimitations

Compiler development is an extensive task and can be continuous as programming languages develop over time. In order to achieve the goals set for this project in the specified time frame, some parts of the compiler were given less attention.

Lexical and syntactic analysis is not a focus of this project. Although important parts to modern compilers, the project will not include the development of custom tools for these parts. Instead, a parser will be generated using the BNF Converter (Abel et al., 2023), which generates a parser according to the grammatical rules specified for the language.

As language design is not at the center of attention in the project, the compiler and its runtime are not meant to compete with other functional programming languages. The resulting language does include common functional features such as lambda functions, pattern matching, and data types. However, these features exist to demonstrate the capabilities of the compiler and not the language itself. Similarly, no extensive standard library will be developed to make the language practical.

1.4 Contributions

This thesis aims to make the following contributions:

- Present Churf, a functional language based on System F.
- Describe the implementation of a Hindley-Milner-based type checker and a bidirectional type checker.
- Describe a methodology for compiling a strict functional language to LLVM IR.
- Demonstrate how to implement a runtime library for a programming language with the purpose of memory management.

2
Theory

This chapter details the necessary theory behind the implementation of Churf, including the formal system lambda calculus and type inference, code generation, and runtime memory management.

2.1 Formal systems & type inference

Following is a presentation of three formal systems; lambda calculus, simply typed lambda calculus, and System F. Additionally, type inference is explained including Hindley-Milner type inference and bidirectional type inference.

2.1.1 Lambda calculus

Lambda calculus is a formal mathematical logic system discovered by Alonzo Church (Church, 1932). The system consists of building terms and performing reductions on them. Following is the syntax of the lambda calculus; the terms are variables, lambda abstractions, and applications.

$$Terms \quad e \quad ::= x \mid \lambda x.e \mid e\,e$$

A variable is bound if the variable is defined locally. For instance, x and y are bound variables in the term $\lambda x.\lambda y.x\,y\,z$ but z is not. A variable that is not bound is called *free*. There are three reductions in lambda calculus: α-conversion, β-reduction, and η-reduction.

α-conversion renames bound variables without changing the semantics of the term. Consider the term $\lambda x.x$. Performing α-conversion on the term using the substitution $[x := y]$ yields $\lambda y.y$. Semantically there is no difference between the terms $\lambda x.x$ and $\lambda y.y$, which means they are α-equivalent. β-reduction evaluates an application by replacing the bound variable in the first term with the second term. The term $(\lambda x.x)(\lambda y.y\,y)$ is β-reduced by the substitution $[x := \lambda y.y\,y]$, which yields $\lambda y.y\,y$. A term is in beta normal form if it cannot be β-reduced further. η-reduction simplifies an abstraction. An abstraction $\lambda x.e\,x$ can be η-reduced if x does not appear as a free variable in e. The resulting equivalent term would be e.

2.1.2 Simply typed lambda calculus

Simply typed lambda calculus restricts lambda calculus by only permitting well-typed terms. The types consist of the type constructor $\rightarrow$ and type variables α from a given set $\mathbb{T}$ (Church, 1940). Curry's style of simply typed lambda calculus uses no type annotations, which means that the terms are identical to the untyped lambda calculus (Curry, 1934; Curry & Feys, 1958; Howard, 1980). In contrast, Church's style of simply typed lambda calculus annotates the bound variable in lambda abstractions $\lambda(x : A).e$. Following is the syntax of the types introduced by the simply typed lambda calculus.

$$\textit{Types} \quad A, B \ ::= A \rightarrow B \mid \alpha \quad \text{where } \alpha \in \mathbb{T}$$

The Curry-Howard correspondence relates the simply typed lambda calculus to intuitionistic logic (Howard, 1980). The correspondence describes the construction and deconstruction of types, for instance, the connective $\rightarrow$ constructs a function. In logic, this corresponds to introduction and elimination rules, and the function constructor $\rightarrow$ corresponds to implication $\implies$.

Type systems can be constructed by combining introduction and elimination rules. For example, abstraction $\lambda x.e$ and application $e_1\, e_2$ can be type checked by the introduction ($\rightarrow$I) and elimination ($\rightarrow$E) rules for $\rightarrow$. Following are the typing rules for simply typed lambda calculus.

$$\frac{}{\Gamma \vdash x : A}\,\text{Var} \qquad \frac{\Gamma, (x : A) \vdash e : B}{\Gamma \vdash \lambda x.e : A \rightarrow B}\,\rightarrow\text{I} \qquad \frac{\Gamma \vdash e_1 : A \rightarrow B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\,\rightarrow\text{E}$$

The judgments use the form $\Gamma \vdash e : A$, read "in context Γ, term e has type A". Γ gives the type of the free variables in e, and $\Gamma, (x : A)$ is the context that extends Γ by associating the variable x with type A. The introduction rule introduces the connective in the conclusion, while the elimination rule eliminates it from the premise. From the rules, the type checking algorithm can be mechanically derived. To check if the abstraction $\lambda x.e$ has type $A \rightarrow B$; extend the context with $x : A$ and check if the proposition $e : B$ holds. Similarly, the application $e_1\, e_2$ has type B if e_1 and e_2 have type $A \rightarrow B$ and A, respectively.

2.1.3 System F

System F is an extension to simply typed lambda calculus that introduces universal quantification over types (Girard, 1986). It was discovered by Girard (1972), and then independently rediscovered by Reynolds (1974). Universal quantification over types, less formally known as parametric polymorphism, enables terms to be defined for several types. The new symbols are type abstractions $\Lambda\alpha.e$, type application $e[A]$, and the polymorphic type $\forall\alpha.A$. Figure 2.1 presents the syntax and typing rules for System F.

$$\begin{array}{lll} \textit{Terms} & e & ::= x \mid \lambda x.e \mid e\,e \mid \Lambda\alpha.e \mid e[A] \\ \textit{Types} & A, B & ::= A \to B \mid \alpha \mid \forall\alpha.A \quad \text{where } \alpha \in \mathbb{T} \end{array}$$

$$\frac{}{\Gamma \vdash x : A}\,\text{Var} \qquad \frac{\Gamma, (x : A) \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\,{\to}\text{I}$$

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\,e_2 : B}\,{\to}\text{E}$$

$$\frac{\Gamma, \alpha \vdash e : A}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.A}\,\forall\text{I} \qquad \frac{\Gamma \vdash e : \forall\alpha.A}{\Gamma \vdash e[B] : A[B/\alpha]}\,\forall\text{E}$$

Figure 2.1: *Syntax and typing rules for the polymorphic lambda calculus System F*

Consider the polymorphic identity function, I, which is written $\Lambda\alpha.\lambda x.x$ and has the type $\forall\alpha.\alpha \to \alpha$. Applying a monomorphic type (A) to the identity function $I[A]$ yields a monomorphic identity function of type $A \to A$. This corresponds to using the $\forall$E rule. It is also possible to apply polymorphic types. For example, instead of $I[A]$ the type application $I[\forall\alpha.\alpha \to \alpha]$ can be used. The result of $I[\forall\alpha.\alpha \to \alpha]$ yields the type $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$. A type system that is able to apply polymorphic types is called impredicative.

To illustrate the benefits of parametric polymorphism, consider it in the context of the Churf program below. There are a few syntactic differences compared to System F. The first difference is top-level definitions, which bind an expression to a variable and has the form `name = e`. Additionally, binds can also have arguments `name x y = e` which is syntactic sugar for `name = \x. \y. e`. Churf uses ASCII, thus symbols such as λ and $\to$ are replaced with `\` and `->`, respectively. Lastly, there are type signatures, which specify a type of a variable or a top-level definition and use the notation `name : A`.

```
identity_int : Int -> Int
identity_int x = x

identity_string : String -> String
identity_string x = x
```

The two functions `identity_int` and `identity_string` have identical bodies, but different type signatures. With parametric polymorphism, it is possible to write one identity function that works for all types.

```
identity : forall a. a -> a
identity x = x
```

2.1.4 Type inference

Static types provide many benefits, however, annotating every term can be considered unusable (Dunfield & Krishnaswami, 2021). Consider a function for integer addition `add : Int -> Int -> Int`. Below is an application of `add` with full type annotations. Type annotations on terms use the notation `e : A`. Notice that each application also needs a type annotation, due to lambda calculus curried functions.

```
four : Int
four = ((add : Int -> Int -> Int) (1 : Int) : Int -> Int) (3 : Int)
```

With type inference, it is possible to deduce a term's type from the context. In this case, assuming the type of `add` is known (`Int -> Int -> Int`), both operands and the returning value must be integers. Thus, it should only be necessary to write `four = add 1 3`.

Type inference combines the convenience of dynamic types with the safety of static types. However, implementing type inference is complicated and sometimes impossible. For instance, full type inference for System F has been proven undecidable (Wells, 1999).

2.1.5 Hindley-Milner type inference

The Hindley-Milner type system was independently discovered by Hindley (1969) and subsequently by Milner (1978). The Hindley-Milner type system builds on two important aspects: the principal type of a term and type inference. A principal type refers to a type for a term such that all other types for that term, in the context, are instances of the principal type. The Hindley-Milner type system is always able to infer the principal type of a term. However, this limits the type system to a subset of System F that is predicative.

An additional construct is introduced in the Hindley-Milner type system, namely, the `let` construct. The following describes the syntax of `let`.

$$\begin{array}{l} \texttt{let}\ x = e \text{ in } e' \\ \texttt{let}\ f\ x_1, ..., x_n = e \text{ in } e' \end{array}$$

Although $\texttt{let}\ x = e\ \texttt{in}\ e'$ and $(\lambda x.e')\ e$ are semantically equivalent, in the Hindley-Milner type system it may be possible to assign correct types to the former whereas the latter would be ill-typed. This is possible because of a concept called let-generalization. For instance, $\texttt{let}\ f = \lambda x.x\ \texttt{in}\ (f\ 5, f\ \texttt{True})$ is well-typed where $\lambda f.(f\ 5, f\ \texttt{True})(\lambda x.x)$ is not. The limitation introduced by not generalizing lambda terms was introduced to aid in proving the soundness theorem for the type system (Milner, 1978).

The type inference algorithm of Hindley-Milner uses the existing unification algorithm described by Robinson (1965). In short, the unification algorithm works as follows. Given a set of constraints, Robinson's unification algorithm will produce a solution in the form of a substitution set if one exists.

Figure 2.2 formally describes the typing rules for instantiation and generalization, as well as the typing rule for the let term.

$$\frac{\Gamma \vdash e : A \qquad A \sqsubseteq B}{\Gamma \vdash e : B}\,\text{Inst}$$

$$\frac{\Gamma \vdash e : A \qquad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall \alpha . A}\,\text{Gen}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x : A \vdash_D e' : B}{\Gamma \vdash \text{ let } x = e \text{ in } e' : B}\,\text{Let}$$

Figure 2.2: *The typing rules introduced in the Hindley-Milner type system*

The typing rule Inst introduces a rule for specializing a type. Consider the two types: A and $\forall a.a$, here $\forall a.a \sqsubseteq A$, meaning $\forall a.a$ is more general than A. Applying the typing rule Inst, $\forall a.a$ can be specialized to A. Gen is the dual of Inst, instead of specializing a type, the typing rule Gen generalizes a type over all bound type variables in the context.

2.1.6 Bidirectional type inference

Bidirectional type checking (Pierce & Turner, 2000) combines two modes: checking and inference. This allows for greater control of what language features need type annotations and which can be inferred. By limiting the amount of inference done, bidirectional type checking remains decidable for very expressive type systems (Abel et al., 2008; Peyton Jones et al., 2007).

Pierce and Turner originally used the name *local type inference* because the technique only uses local information. As a result, it is easier to implement high-quality error messages, compared to unification-based algorithms such as Hindley-Milner (Milner, 1978).

The combination of checking and inference means that there are multiple valid rules for each term, and it is not obvious how to determine which is best. Fortunately, Dunfield and Krishnaswami (2021) have defined four criteria for assessing a bidirectional typing system:

(1) Mode-correctness: types should never be guessed

(2) Completeness: all terms match at least one of the rules

(3) Size: fewer rules are easier to work with

(4) Annotation character: annotations should be lightweight, predictable, stable, and legible

Dunfield and Krishnaswami also presented a method for creating such a typing system, which is an alteration of the *Pfenning recipe* by Dunfield and Pfenning (2004). The most important component of the recipe is the treatment of the principal judgment, which is the rule containing the principal connective. In most cases, the principal judgment is the conclusion for an introduction rule and one of the premises for an elimination rule. The key to the recipe is making the principal judgment either checking or inferring depending on the mode of the rule; checking for introduction and inferring for elimination. Then, the mode of the remaining judgments is usually easy to determine.

The subsumption rule, Sub, is the key to fulfilling the third assessment criterion. Subsumption is a checking rule that infers a type A and then checks if it is equal to the supplied type B (Dunfield & Krishnaswami, 2021). This is possible due to checking containing more information (the supplied type) than inference. Therefore, switching from inference to checking is not possible as the supplied type is missing.

$$\frac{\Gamma \vdash e \downarrow A \qquad A = B}{\Gamma \vdash e \uparrow B}\ \text{Sub}$$

Dunfield and Krishnaswami (2021, 2013) use the notation $\Gamma \vdash e \Leftarrow A$ and $\Gamma \vdash e \Rightarrow A$ for checking and inference judgements, respectively. However, this thesis uses the notation $\Gamma \vdash e \uparrow A$ and $\Gamma \vdash e \downarrow A$ similar to (Norell, 2007). This is in order to avoid confusion due to the similarity of inference and the branch syntax $p \Rightarrow e$.

2.2 Code Generation

Code generation is usually one of the last phases in a compiler. Given an AST, the code generator will translate the tree into machine code, or assembly code which in turn can be compiled into machine code (Ranta, 2012). The most important aspect of this phase is that the code produced is correct and represents the original program (Aho et al., 2007).

An assembly language is a low-level programming language that closely represents CPU instructions. These languages consist of opcodes that represent machine language instructions, labels, and in some cases, more advanced features like macros (Streib, 2020).

Each CPU architecture typically have accompanying assembly-languages making each assembly language inherently CPU-architecture dependent. For a program to be compatible with multiple architectures, code generation for more than one assembly language is necessary.

2.2.1 LLVM

The creation of LLVM started in 2000, and while it was initially developed to explore multi-stage optimization (Lattner, 2002), it has over time evolved to be a relatively large compiler framework encompassing many different smaller projects. One of these projects is the intermediate representation assembly language LLVM IR, which is a high-level assembly language which can be compiled to many different assembly languages, leading to the support of many different CPU-architectures (LLVM Developer Group, 2023a).

This intermediate language closely resembles the lower-level assembly languages, while also introducing more modern features, such as a type system, type declarations, function definitions, and more (LLVM Developer Group, 2023c).

LLVM IR is written in Static Single Assignment form (SSA), which introduces the rule that variables may only be assigned once. This guarantee makes optimization of the IR code easier, as unused and unnecessary variables can easily be optimized away, which is useful in many optimization algorithms (LLVM Developer Group, 2023d).

2.3 Runtime

A runtime can provide useful functionality such as concurrency, heap allocations, and exception handling. These features can be interfaced with through a runtime library. This chapter focuses on functionality used for managing heap memory, which is handled by a language's runtime.

2.3.1 Memory

Memory is a shared resource in modern operating systems, therefore it is important to manage it over several processes (Silberschatz et al., 2018). For a given process, its assigned memory consists of different sections. Two sections are the stack and the heap. A process's stack contains variables limited to function scopes and function calls represented by stack frames. The size of stack data is often limited as the stack serves multiple purposes, such as containing register values and temporary data. The heap contains explicitly allocated memory used for dynamic data. During the execution of the program, these two sections grow towards each other, as seen in figure B.1 in the appendix. The stack is a contiguous part of memory and is directly managed by the compiled program whereas the heap requires the program to request segments from the operating system.

2.3.2 Managing heap memory

In many modern high-level languages, heap memory is automatically managed by its runtime. Runtime libraries consist of crucial functionality to programming languages. The size of runtime libraries varies in size depending on the goal of the language. For example, the runtime library of C is small in comparison to Java's runtime library.

In many popular high-level languages, garbage collection is included as a part of the language's runtime library. Without a garbage collector, a process can request heap memory but also has to explicitly request to free it. Failing to free allocated memory can lead to memory leaks and unnecessarily high memory usage. Excessive use of heap memory can lead to heap exhaustion and the process running out of memory (Jones et al., 2012). Garbage collection is the method of preventing these consequences by freeing memory that is no longer used. It guarantees that allocated but unused heap memory is eventually freed and returned to the process heap.

2.3.3 The Mark-Sweep algorithm

The first garbage collection algorithm was the Mark-Sweep algorithm in the early 60s (McCarthy, 1960). Mark-Sweep is known as a *stop-the-world* algorithm, where the process execution is paused while the garbage collector analyzes the process memory (Jones et al., 2012). During analysis, the algorithm determines what memory is unused and frees it. The Mark-Sweep algorithm is split into two phases, marking and sweeping. The marking phase traverses a graph of all objects reachable from the current state of process memory and marks them. As the set of marked memory is a subset of all allocated memory, the sweeping phase can then iterate over all allocated memory and free unmarked memory.

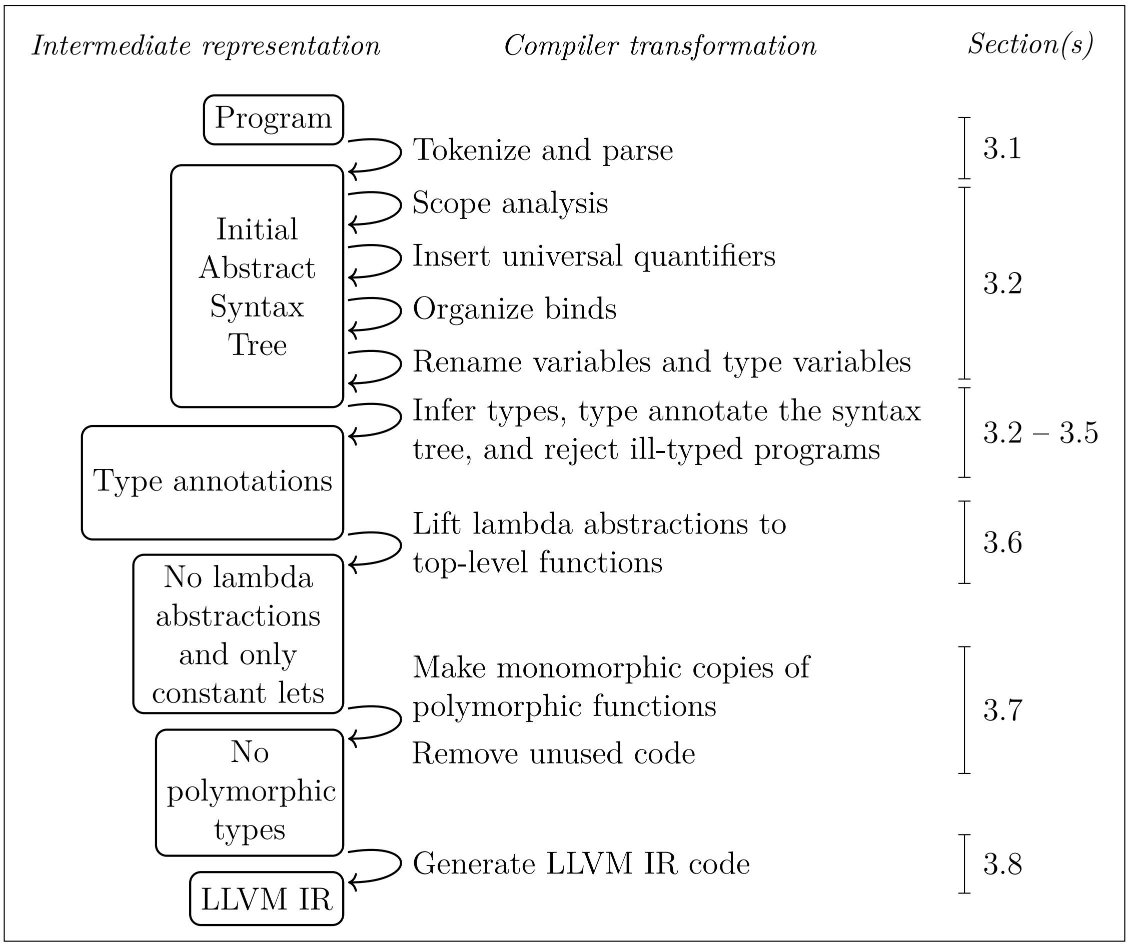
The Mark-Sweep algorithm can be implemented as a conservative garbage collector (Jones et al., 2012). A conservative collector assumes that some data is a pointer if it looks like one (Jones et al., 2012). If the decision to distinguish pointers is not accurate, there is a possibility of memory being freed too early or not at all, which breaks the purpose of garbage collection. Hence, the set of actual pointers in memory might only be a subset of the pointers decided by the collector. This guarantees that no pointer will be missed during marking, though more data is analyzed than necessary which impacts performance.

Since the analyzed data include non-pointers, the collector cannot alter or move any data (Jones et al., 2012). This rules out the possibility of moving allocated memory for space efficiency. Therefore, memory allocated sequentially might become fragmented with empty gaps between used memory. These gaps can be of any size, but smaller gaps cannot meet allocation requests of the same size as larger gaps can. If these gaps are small and frequent enough, the heap may be exhausted prematurely, since the gaps are not contiguous they cannot satisfy allocation requests. This is called memory fragmentation and all conservative collectors suffer from it to some degree (Jones et al., 2012).

3

Compiler

The compiler consists of several phases for analysis and transformation. This chapter describes each phase. An overview of the compiler is depicted in figure 3.1. The first column describes the intermediate representations, in the form of an abstract syntax tree (AST), used when transforming a program into LLVM IR. Next, the adjacent arrows represent compiler transformations. Lastly, the corresponding section(s) of each transformation is displayed in the rightmost column.



Intermediate representation
Compiler transformation
Section(s)
Program
Tokenize and parse
3.1
Initial Abstract Syntax Tree
Scope analysis
Insert universal quantifiers
Organize binds
Rename variables and type variables
3.2
Infer types, type annotate the syntax tree, and reject ill-typed programs
3.2 – 3.5
Type annotations
Lift lambda abstractions to top-level functions
3.6
No lambda abstractions and only constant lets
Make monomorphic copies of polymorphic functions
Remove unused code
3.7
No polymorphic types
Generate LLVM IR code
3.8
LLVM IR

Figure 3.1: *Overview of the Churf compiler.*

3.1 Parsing the program

The parser is the entry point of the compiler. It transforms the program into the initial AST via two phases; lexical analysis and syntactic analysis. In the first phase, the lexer splits the program into a stream of tokens. For example, `f = \x. x + x` is split into: `"f"`, `"="`, `"\"`, `"x"`, `"."`, `"x"`, `"+"`, `"x"`, `";"`. Then, the parser converts the tokens into the syntax tree, according to rules specified by the context-free grammar, as seen in figure 3.2.

```
Program [DBind (Bind (VIdent (LIdent "f")) []
  (EAbs (LIdent "x")
      (EApp
          (EApp
              (EVar (VSymbol (Symbol "+")))
              (EVar (VIdent (LIdent "x"))))
          (EVar (VIdent (LIdent "x"))))))]
```

Figure 3.2: *Initial AST for the program* `f = \x. x + x`.

Both the lexer and parser are generated by the BNF Converter (Abel et al., 2023), which accepts a grammar written in BNF notation. A stylized form of the Churf grammar is specified in figure 3.3. The grammar is similar to that of Haskell, including the option to mix indentation-sensitive code with code using braces and semicolons. Additionally, it is possible to define infix functions, such as the addition operator `+`. However, this is only possible for symbols.

3.2 Renaming and rearranging definitions

Renaming and rearranging definitions is the beginning of the semantic analysis phase. Two goals of the renaming and rearranging definitions phase are to catch errors that are not syntactic nor type related and to prepare the AST for the type checker.

A common category of errors is scoping errors, which occur when parts of a program try to access variables that are not defined or out of scope. Programs are also rejected when higher-rank polymorphism (see section 3.5) is used with the Hindley-Milner type checker. The last step is reporting uses of redundant universal quantifiers, for instance, `identity : forall a. forall b. a -> a`.

There are advantages and disadvantages to modifying the AST prior to the type checker; type checking becomes easier but error messages suffer. For that reason, this phase does not create a new AST and only makes small modifications. The first modification is renaming (α-conversion) of variables and type variables which means that the type checker does not have to consider naming-related scoping issues. Another modification is surrounding types containing unbound type variables with universal quantifiers. This makes it possible to write rank-1 polymorphism without

Meta variables

x, y	Term-variable name
lit	Literal
K	Data-constructor name
α, β	Type-variable name
T	Type-constructor name

Grammar

$Program$	$program$	$::=$	$\overline{def}$	`Program`: Program

Def	def	$::=$	$x\,\overline{x} = e$	`DBind`: Binding
		$\mid$	$x : A$	`DSig`: Signature
		$\mid$	$\mathrm{data}\,A\,\mathrm{where}\,\overline{k : A}$	`DData`: Data type

$Terms$	e	$::=$	x	`EVar`: Variables
		$\mid$	$\lambda x.e$	`EAbs`: Lambda abstraction
		$\mid$	$e\,e$	`EApp`: Application
		$\mid$	$\mathrm{let}\,x\,\overline{x} = e\,\mathrm{in}\,e$	`ELet`: Let-bindings
		$\mid$	$e : A$	`EAnn`: Type annotated expression
		$\mid$	lit	`ELit`: Literal
		$\mid$	K	`EInj`: Data type constructor
		$\mid$	$\mathrm{case}\,e\,\mathrm{of}\,\overline{p \Rightarrow e}$	`ECase`: Case expression

$Patterns$	p	$::=$	x	`PVar`: Variable
		$\mid$	lit	`PLit`: Literal
		$\mid$	K	`PEnum`: Data constructor with no arguments
		$\mid$	$K\,p\,\overline{p}$	`PInj`: Data constructor with pattern arguments
		$\mid$	$\_$	`PCatch`: Wildcard

$Types$	A, B	$::=$	$A \to B$	`TFun`: Function
		$\mid$	α	`TVar`: Type variable
		$\mid$	$\hat{\alpha}$	`TEVar`: Type existential variable (internal)
		$\mid$	$\forall \alpha.A$	`TAll`: Polymorphic type
		$\mid$	$T\,\overline{A}$	`TData`: Data type with parameters

Figure 3.3: *Formal grammar of the Churf language. Each production has a corresponding data constructor in the initial AST. For example, the data constructor of $\lambda x.e$ is `EAbs`. Also, the overline represents a sequence of zero or more. Churf programs use similar grammar other than the Unicode symbols, which are replaced by an ASCII alternative.*

quantifiers, for example, `identity : a -> a`. Finally, top-level binds are ordered based on usage to make type inference easier. The ordering is similar to a topological sort; binds that reference another bind are placed below the reference. However, one crucial difference is that binds can be mutually recursive, which means that the graph might be cyclical.

3.3 Type checking

Type checking is an important part of the semantic analysis. Its purpose is to reject ill-typed programs and to annotate the AST with types. The type checker receives a program that may include some type annotations or type signatures, but the remaining types must be inferred. Furthermore, the previous phase ensures that every polymorphic type present in the input has explicit universal quantifiers, and all the variable names are unique.

Following is an example of the transformation by the type checker. In figure 3.4, the function `const` consists only of a type signature which means that it will type check, but will be rejected in later phases of the compiler.

```
-- Input
const : forall $0a. forall $1b. $0a -> $1b -> $0a

main = const 65 'a'

-- Output
const : forall $0a. forall $1b. $0a -> $1b -> $0a

(main : Int) = (((const : Int -> Char -> Int)
                (65 : Int) : Char -> Int) ('a' : Char) : Int)
```

Figure 3.4: *An example of the transformation from the type checker phase.*

There are two type checkers in the Churf compiler with different capabilities. The first type checker is based on the Hindley-Milner type system, and it is able to infer all types of a program without requiring any type annotations. In contrast, the second type checker uses a bidirectional approach and requires some type annotations. However, in return, the bidirectional type checker is able to accommodate higher-rank polymorphism.

In the two following sections, the ideas and implementation of the two type checkers are described. However, detailed implementations of the type checkers are beyond the scope of this thesis. Curious readers are instead encouraged to read the original sources; Milner (1978), Dunfield and Krishnaswami (2013), respectively.

Another aspect that is not detailed in the following sections is type checking data types and pattern matching. These primitives are not included in System F, however, they increase the usefulness of the language by enabling data structures such as lists and trees. Data types are defined by the keywords `data ... where` and can be pattern matched by `case ... of`. The current implementation is able to type check nested pattern matching. However, this feature is not yet implemented in the backend of the compiler and is as a result unusable.

3.4 Hindley-Milner type checker

The Hindley-Milner type system can be implemented using one of two algorithms, Algorithm W and Algorithm J. The former of which as described by Milner is side effect free whereas the latter is not (Milner, 1978). Hence, Algorithm W is more appropriate for a pure functional language.

Figure 3.5 contains the typing rules for Algorithm W. The notation $\Gamma \vdash e : A, S$ is a relation between four parts: a context Γ, a term e, a type A, and a substitution S. The rules can be seen as a logic program with Γ and e as inputs. If it succeeds, the output is the type of the term A and the substitution S.

$$\frac{x : A \in \Gamma \qquad B = \mathit{inst}(A)}{\Gamma \vdash x : B, \emptyset}\ \text{Var}$$

$$\frac{\begin{array}{cc}\Gamma \vdash e_0 : A, S_0 & S_0\Gamma \vdash e_1 : B, S_1 \\ C = \mathit{fresh}(\Gamma) & S_2 = \mathit{unify}(S_1A, B \to C)\end{array}}{\Gamma \vdash e_0 e_1 : S_2C, S_2S_1S_0}\ \text{App}$$

$$\frac{A = \mathit{fresh}(\Gamma) \qquad \Gamma, x : A \vdash e : B, S}{\Gamma \vdash \lambda x.e : SA \to B, S}\ \text{Abs}$$

$$\frac{\Gamma \vdash e_0 : A, S_0 \qquad S_0\Gamma, x : \overline{S_0\Gamma}(A) \vdash e_1 : B, S_1}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : B, S_1S_0}\ \text{Let}$$

Figure 3.5: *Typing rules for Algorithm W in the Hindley-Milner type system*

Algorithm W additionally introduces the generation of fresh variables ($\mathit{fresh}(\Gamma)$), substitutions (S_n), applying substitutions to types (S_nA) and contexts ($S_n\Gamma$), and composing substitutions ($S_nS_{(n-1)}$). The composition order of substitutions matters as cases may arise where substitutions are reduced too early or too late. The remaining parts of the section present Algorithm W and its implementation in Churf.

When unifying a type variable, defined as `TVar` in the initial AST (see figure 3.3), with any other type, the work is deferred to the `occurs` function. `occurs` checks if the type variable occurs as a free type variable in the other type. For instance, $unify(\alpha, \alpha \to \beta)$ would fail because the type variable α appears as a standalone type variable in the first argument, and as a type variable in the function type $\alpha \to \beta$ in the second argument. If the occurs-check succeeds, the singleton substitution containing the mapping from the type variable to the type is returned.

```
unify :: Type -> Type -> Except String Substitution
unify (TVar var) t = occurs var t
unify t (TVar var) = occurs var t
```

A substitution is defined as a set of mappings from the identifier of a type variable to a type.

```
data Substitution = Map Ident Type
```

The unification of two type literals is easier, as the only check needed is to see if the literals are equal. If the literals are equal, return the empty substitution, otherwise, report an error.

```
unify (TLit lit_left) (TLit lit_right)
     | lit_left == lit_right = emptySubsitution
     | otherwise = throwError "Unification of type literals failed"
```

Lastly, there is the function type. The left side of the function types are unified recursively. The substitution produced by the recursive unification is then applied to the right side of both function types, which in turn are recursively unified as well. The two substitutions are then composed into one.

```
unify (TFun t0 t1) (TFun t2 t3) = do
    sub_0 <- unify t0 t2
    sub_1 <- unify (apply sub_0 t1) (apply sub_1 t3)
    return (sub_0 `compose` sub_1)
```

Type inference for most Churf terms is implemented using the aforementioned typing rules. The algorithm pattern matches on each possible term in the initial syntax and handles each case differently.

For instance, the typing rule Abs generates a fresh type variable, binds the fresh type to the bound variable of the abstraction, then continues inferring the type of the inner term.

The algorithm as seen in figure 3.6 then returns the function type from the fresh type variable to the inferred type, where the substitution is applied to the fresh type variable. This process is depicted in `let funType = ...` in figure 3.6.

A fresh type variable is a type variable that does not occur anywhere else in the term. The term `EVar` correspond to the `Var` typing rule, however, a slight addition

```
algorithm_w :: Exp -> Except String (Substitution, Exp, Type)
algorithm_w (EAbs name expr) = do
    freshTypVar <- fresh
    withBinding name freshTypVar $ do
        (subst, expr_inferred, typInf) <- algorithm_w expr
        let funTyp = TFun (apply subst freshTypVar) typInf
        return (subst, EAbs name expr_inferred, funTyp)
```

Figure 3.6: *Implementation of lambda abstractions in Algorithm W*

in Churf is that an `EVar` can not only refer to a variable in a term, but also a bind declared at the top level. Thus, the algorithm as seen in figure 3.7 first checks if the identifier is declared in the current context, if it is, instantiate the type and return it, as well as the empty substitution set. However, if the identifier is not found as a declared local variable then the algorithm checks if it is a declared bind. Since the Hindley-Milner type system can infer type signatures for binds, two cases arise: if the bind is declared with a signature, then the function returns that type with the empty substitution set. If the bind is declared without a type signature, then assign a fresh type variable to the bind ready for unification at a later stage. If the identifier is neither found as a declared local variable nor as a bind, then an unbound variable error is reported.

```
algorithm_w (EVar identifier) =
    case lookup identifier declaredVars of
        Just t -> do
            t' <- instantitate t
            return (nullSubst, EVar i, t')
        Nothing ->
            case lookup identifier declaredSignatures of
                Just (Just t) -> return (nullSubst, EVar i, t)
                Just Nothing -> do
                    freshTypVar <- fresh
                    return (nullSubst, EVar i, freshTypVar)
                Nothing -> throwError ("Unbound variable: " ++ i)
```

Figure 3.7: *Implementation of variables in Algorithm W*

Churf allows type annotations on terms. For instance, the term `\x. x` could be annotated as `\x. (x : Int)`, thus specifying exactly what its type should be. Although type annotations are not explicitly specified by the Hindley-Milner type system, the simplicity of the type system enables extensions such as type annotations easily. The implementation of type inference for annotated types is described by the code in figure 3.8.

```
algorithm_w(EAnn expr typAn) = do
    (sub_0, expr_inferred, typInf) <- algorithm_W e
    sub1 <- unify typAn typInf
    unless (typInf <<= typAn) (throwError "Error")
    return (sub_1 `compose` sub_0, expr_inferred, typAn)
```

Figure 3.8: *Implementation of type annotated expressions in Algorithm W*

The inferred type of the term is compared to the annotated type. As long as the inferred type is more general than the annotation, which is checked by `<<=`, it is accepted. The function `<<=` thus correspond to $\sqsubseteq$ in the typing rule for Inst.

3.5 Bidirectional type checker

The bidirectional type checker is an implementation of the algorithmic typing system presented in *Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism* by Dunfield and Krishnaswami (2013). The implementation adjusts and extends the typing system to fit the Churf specification. Some of the additions are let-bindings (both local and top-level), literals, and algebraic data types.

Higher-rank polymorphism allows universal quantifiers to appear unrestricted in a type. Comparatively, Hindley-Milner only allows quantifiers on type schemes, meaning that the quantifier can only appear on the outside (Dunfield & Krishnaswami, 2021). For example, $\forall\alpha.(\forall\beta.\beta \to \beta) \to \alpha \to \alpha$ and $\forall\alpha.\alpha \to (\forall\beta.\beta \to \beta) \to \alpha$ are higher rank but $\forall\alpha.\forall\beta.\alpha \to (\beta \to \beta) \to \alpha$ is not. One interesting application of higher-rank polymorphism is that it enables monads to be defined with regular algebraic data types (Peyton Jones et al., 2007; Wadler, 1990).

Consider the expression `\x.x+x`. As mentioned in section 2.1.2, type-checking a lambda abstraction is easy. Extend the context by associating x with type A, and check that the body is of type B.

$$\frac{\Gamma, (x : A) \vdash e \uparrow B}{\Gamma \vdash \lambda x.e \uparrow A \to B} \to\mathrm{I}$$

Type inference is considerably more difficult. Like many functional programs, the type checker starts with the outermost expression and then recurses into the subexpressions. This is problematic for inferring the lambda abstraction because the type of x (A) is unknown when extending the context. Dunfield and Krishnaswami solves this problem with an ordered context, combined with type existential variables ($\hat{\alpha}$) and scoping markers ($\blacktriangleright_{\hat{\alpha}}$). Moreover, every judgment has an output context Δ.

$$\Gamma, \Delta, \Theta ::= \cdot \mid \Gamma, \alpha \mid \Gamma, (x : A) \mid \Gamma, \hat{\alpha} \mid \Gamma, (\hat{\alpha} = \tau) \mid \Gamma, \blacktriangleright_{\hat{\alpha}}$$

Using this solution, the type of `\x.x+x` can be inferred by extending the context with placeholder types $\hat{\alpha}$, $\hat{\beta}$, and the type assignment $x : \hat{\alpha}$. Then, the placeholder values are hopefully solved ($\hat{\alpha} = \mathrm{Int}, \hat{\beta} = \mathrm{Int}$) during checking of the body e.

$$\frac{\Gamma, \hat{\alpha}, \hat{\beta}, (x : \hat{\alpha}) \vdash e \uparrow \hat{\beta} \dashv \Delta, (x : \hat{\alpha}), \Theta}{\Gamma \vdash \lambda x.e \downarrow \hat{\alpha} \to \hat{\beta} \dashv \Delta} \to\mathrm{I}\!\downarrow$$

Checking will fail if the lambda abstraction is a polymorphic function, for instance, `\x. x`. The typing system requires annotations on polymorphic functions to allow higher-rank polymorphism `\x. x : forall a. a -> a`. Another compromise is impredicative inference (see section 2.1.3).

The typing rules are available in figure A.1 in the appendix. Most of the rules are identical to the rules defined by Dunfield and Krishnaswami (2013). Some of the additions to the system are listed below. Moreover, some implementation-specific changes are not visible in the typing system (figure A.1) but are mentioned in the remaining parts of this section.

- Lit are Lit$\downarrow$ are two new rules for checking and inference of literals.

$$\frac{\text{typeof}(\mathit{lit}) = A}{\Gamma \vdash \mathit{lit} \uparrow A \dashv \Gamma}\ \text{Lit} \qquad \frac{}{\Gamma \vdash \mathit{lit} \downarrow \text{typeof}(\mathit{lit}) \dashv \Gamma}\ \text{Lit}\downarrow$$

- VarRec is a new rule that type check recursive function calls. The rule is simple; if the variable is recursive then the context is extended by associating the variable with a fresh type existential variable.

$$\frac{\Gamma \vdash \mathit{rec}(x)}{\Gamma \vdash x \downarrow \hat{\alpha} \dashv \Gamma, (x : \hat{\alpha})}\ \text{VarRec}$$

- Let-terms are an addition to the type system, and they are type checked in two steps. First, the bind is inferred by converting it to a lambda abstraction. Then, the output context is extended with the name of the bind associated with the inferred type, and the type of resulting term (e') is inferred. Lastly, the output context excludes the variable assignment ($x : A$).

$$\frac{\Gamma \vdash \lambda\overline{y}.e \downarrow A \dashv \Theta_1 \qquad \Theta_1, (x : A) \vdash e \downarrow B \dashv \Delta, (x : A), \Theta_2}{\Gamma \vdash \text{let } x\,\overline{y} = e \text{ in } e' \downarrow B \dashv \Delta}\ \text{Let}$$

The most challenging part of implementing the typing system was propagating the typing information of the context to the AST. Dunfield and Krishnaswami uses the notation $[\Theta]A$ for context application, which ensures that A does not contain any type existential variables that are solved in the context Θ. Context application ($[\Theta]A$) is similar to applying a substitution to a type in unification-based inference, such as Hindley-Milner. Context application is routinely used in the typing system for rules with multiple premises. One example is inference for application $\to$E shown below. The judgment form $\Gamma \vdash A \bullet e \Downarrow C \dashv \Delta$ is called application, and it is a special case of inference that also performs type application on polymorphic functions.

$$\frac{\Gamma \vdash e_1 \downarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \bullet e_2 \Downarrow C \dashv \Delta}{\Gamma \vdash e_1\ e_2 \downarrow C \dashv \Delta}\ \to\text{E}$$

The solution to the problem of propagating type information to the AST is to enforce a new post-condition that all output types need to be fully applied to the output context. Below is the implementation of $\to$E. The functions `infer`, `applyInfer`, and `apply` correspond to the respective judgments: inference, application, and context application. There are some changes compared to the rule $\to$E. The output of `infer` and `applyInfer` is different as it outputs a type-annotated expression along with the inferred type. Additionally, `applyInfer` also outputs the function type after the type application. Another change is the context application which is performed as the last operation. This change is a result of the new post-condition, which in turn makes context application prior to `applyInfer` redundant.

```
-- rule: →E
infer :: Exp -> State Cxt (T.Exp, Type)
infer (EApp e1 e2) = do
    (e1', a) <- infer e1
    (e2', a', c) <- applyInfer a e2
    apply (T.EApp (e1', a') e2', c)
```

Free variables	Term
∅	`(\x.\ys. map (\y. y + x) ys) 4 (Cons 1 (Cons 2 Nil))`
∅	`(\x.\ys. map (\y. y + x) ys) 4`
∅	`\x.\ys. map (\y. y + x) ys`
x	`\ys. map (\y. y + x) ys`
x, ys	`map (\y. y + x) ys`
x	`map (\y. y + x)`
∅	`map`
x	`\y. y + x`
y, x	`y + x`
y	`y +`
∅	`+`
y	`y`
x	`x`
ys	`ys`
∅	`4`
∅	`Cons 1 (Cons 2 Nil)`
∅	`Cons 1`
∅	`Cons`
∅	`1`
∅	`Cons 2 Nil`
∅	`Cons 2`
∅	`Cons`
∅	`2`
∅	`Nil`

Figure 3.9: *Free variables of each term.*

3.6 Lifting lambda functions

Lambda lifting is a technique for converting lambda functions into ordinary top-level functions (Johnsson, 1985). The implementation is a modified version of the algorithm presented in chapter 6 of *Implementing Functional Languages: A Tutorial* (Peyton Jones & Lester, 1992). The algorithm has three phases: annotating free variables, replacing lambda abstractions with let terms, and lifting let terms into top-level functions. As an example, consider the following program[1]:

```
f : List Int
f = (\x.\ys. map (\y. y + x) ys) 4 (Cons 1 (Cons 2 Nil))
```

There are three lambda abstractions in the program that need to be lifted. The first phase is to annotate each term with its free variables. For example, `x` is the only free variable in `\y. y + x`. Only locally defined free variables are annotated as global variables are accessible from most contexts. The results of the first phase are displayed in figure 3.9. Next, lambda abstractions are replaced with let-terms, and successive lambda abstractions are combined.

[1]Type annotations from the type checker are omitted for readability.

```
f = let
       sc_0 x ys = map (let {x̄}sc_1 y = y + x̄ in {x̄:=x}s_1) ys
     in
       sc_0 4 (Cons 1 (Cons 2 Nil))
```

The transformation varies depending on whether the lambda abstraction contains free variables. Self-contained lambda abstractions are replaced with ordinary binds, whereas lambda abstractions with free variables (closures) are replaced with a bind with an associating context. In the example, `sc_0` is the former transformation, and `sc_1` is the latter.

Lastly, the binds are lifted to the top level. The resulting AST has no lambda abstractions and let-terms can only be constants. Constant let-terms are allowed as they are equivalent to local variables in LLVM IR. Additionally, the phase introduces two new constructions; a bind with an associated context `{x̄}sc = ...` and context instantiation `{x̄:=x}sc`.

```
f = sc_0 4 (Cons 1 (Cons 2 Nil))
sc_0 x ys = map {x̄:=x}sc_1 ys
{x̄}sc_1 y = y + x̄
```

3.7 Monomorphization

LLVM handles computer resources directly, which requires known sizes of data such as function arguments and types in data structures. Polymorphic types do not have fixed sizes and need to be handled before code generation. One approach is boxing, which involves pointers being passed between functions instead of values, which with boxing are stored on the heap. Since pointers always have the same size, the sizes of arguments are always known. One downside of boxing is unboxing, the process of accessing the pointers' memory and reading or writing to the underlying data. Reading the pointer does take time which is not desired.

Trading faster run times for longer compile times, monomorphization is the chosen method for handling polymorphic types in Churf. Monomorphization is the process of converting polymorphic functions and data types to a set of identical definitions using concrete monomorphic types. While monomorphization avoids some runtime costs, the fact that multiple definitions of the same function are generated can increase the compiled file's size.

Before monomorphization, every term in the AST has type annotations which include polymorphic types. Polymorphic functions are monomorphized based on their usage. When a polymorphic function is applied to a monomorphic value, the monomorphization algorithm creates a monomorphic copy. `main` is the entry point to every program, thus all functions but those indirectly used by `main` will be removed. The following is an example that monorphizes the `const` function into two different monomorphic declarations. Note that because of readability, application and addition are not explicitly annotated with types. Additionally, the generated function identifiers do not represent those used in the Churf compiler.

```
const : forall $0a. forall $1b. $0a -> $1b -> $0a
const (x : $0a) (y : $1b) = (x : $0a)

main =
    (const : Int -> Int -> Int)
    (10 : Int) (10 : Int) +
    (const : Int -> Char -> Int)
    (65 : Int) ('a' : Char)
```

After monomorphization:

```
const_1: Int -> Int -> Int
const_1 (x : Int) (y : Int) = (x : Int)

const_2 : Int -> Char -> Int
const_2 (x : Int) (y : Char) = (x : Int)

main =
    (const_1 : Int -> Int -> Int) (10 : Int) (10 : Int)
    + (const_2 : Int -> Char -> Int) (65 : Int) ('a' : Char)
```

Monomorphization of data types resembles that of functions but possesses the property that some types become the monomorphic type `Void`. In the following example, the constructor `Right` holds a value of type `b`. The constructor is never used which makes monomorphization of `b` impossible, which can be seen as `Right` is not included in the resulting AST. This results in `b` becoming the monomorphic type `Void`, which can not have associated values. `Void` types only appear in the AST after unsuccessful monomorphizations which only happen on types that have no associated values, hence `Void` is never needed and poses no problem in code generation.

```
data Either a b where
    Left  : a -> Either a b
    Right : b -> Either a b

main : Int
main = case (Left 5) of
    Left x => x
```

```
data Either Int Void where
    Left : Int -> Either Int Void

main : Int
main = case (Left 5) of
    Left x => x
```

3.8 Code generation

The last phase of the compiler is generating LLVM IR code. According to LLVM Developer Group (2023b), there are three approaches to utilizing LLVM; use the official LLVM C library, generate LLVM bytecode manually, or generate LLVM IR code manually. This project uses the last alternative, as generating bytecode would lead to making debugging harder during development. While there are bindings to the C library in Haskell, a majority of them only target older versions of LLVM and GHC, at least at the time of writing.

The implementation of the code generator is based on the algorithms described in *Implementing Programming Languages* (Ranta, 2012), specifically, the code generator targeting the Java Virtual Machine in chapter 6, and the interpreter for a functional language outlined in chapter 7.

For the most part, translating the AST into LLVM IR works well due to the immutable nature of lambda calculus and the SSA form of LLVM IR. In the remaining parts of this section, the translation to LLVM IR is demonstrated including the translation of functions, data types, and pattern matching.

3.8.1 Translating functions

To illustrate the translation of functions, consider the program from section 3.6:

```
f = sc_0 4 (Cons 1 (Cons 2 Nil))
sc_0 x ys = map {x̄:=x}sc_1 ys
{x̄}sc_1 y = y + x̄
```

The first function, `f`, is a series of applications that returns a list of integers. In LLVM IR, 64-bit integers have the type `i64` and lists are represented with a custom struct type `%List`, which is described in section 3.8.2. Function application uses the instruction `call` together with the return type of the function and the function identifier with its arguments. In the translation of `f` in figure 3.10, the function signature and every function call include an argument of the pointer type `ptr`. This is in order to handle functions with a context of free variables, such as `{x̄}sc_1`.

```
define %List @f(ptr %cxt) {
    %1 = call %List @Nil(ptr null)
    %2 = call %List @Cons(ptr null, i64 2, %List %1)
    %3 = call %List @Cons(ptr null, i64 1, %List %2)
    %4 = call %List @sc_0(ptr null, i64 4, %List %3)
    ret %List %4
}
```

Figure 3.10: *The translation of* `f = sc_0 4 (Cons 1 (Cons 2 Nil))` .

Before `{x̄}sc_1`, consider the function that creates the context of free variables: `sc_0`. Figure 3.11 displays the translation of `sc_0`. This translation is tricky because `map` is a higher-order function, and the function argument contains a free variable which is defined in `sc_0`. The first step is to create a context type `%Cxt_sc_1` containing the type of the function (`i64 (ptr, i64)*`) and the type of the free variable (`i64`). Next, a structure of the context type is allocated on the stack, and then both the function `@sc_1` and the free variable `%x` are stored at the appropriate indexes. Allocation uses the instruction `alloca`, and variables are stored using `store`. To calculate the correct index, the `getelementptr` instruction is used.

The last function is `{x̄}sc_1`, which adds the variable `y` with the free variable `x̄`. As depicted in figure 3.12, the value of the free variable is retrieved from the context using `load`. Finally, the two variables are added by the instruction `add`.

```
%Cxt_sc_1 = type { i64 (ptr, i64)*, i64 }

define %List @sc_0(ptr %cxt, i64 %x, %List %ys) {
    %sc_1 = alloca %Cxt_sc_1
    %1 = getelementptr %Cxt_sc_1, ptr %sc_1, i32 0, i32 0
    store i64 (ptr, i64)* @sc_1, ptr %1
    %2 = getelementptr %Cxt_sc_1, ptr %sc_1, i32 0, i32 1
    store i64 %x, ptr %2
    %3 = call %List @map(ptr null, ptr %sc_1, %List %ys)
    ret %ListInt %3
}
```

Figure 3.11: *The translation of* `sc_0 x ys = map {x̄:=x}sc_1 ys` .

```
define i64 @sc_1(ptr %cxt, i64 %y) {
    %1 = getelementptr %Cxt_sc_1, ptr %cxt, i32 0, i32 1
    %x = load i64, ptr %1
    %2 = add i64 %y, %x
    ret i64 %2
}
```

Figure 3.12: *The translation of* `{x̄}sc_1 y = y + x̄` .

3.8.2 Translating data types

Some data types are already present in LLVM IR, and this is the case for the built-in Churf types `Int`, `Char`, as well as the data type `Bool`. These types are represented using their respective LLVM IR types, `i64`, `i8`, and `i1`. However, data types other than `Bool` require special treatment, and they have to be implemented using the LLVM IR's `type` keyword. The program which has been used for the examples has used a data type `List`:

```
data List where
    Cons : Int -> List -> List
    Nil  : List
```

This linked list data type contains two data constructors, `Cons` which holds an integer and tail of the list, and `Nil` which is the empty list. This data type will be translated to the following LLVM IR code:

```
%List = type { i8, [23 x i8] }
%Cons = type { i8, i64, %List* }
%Nil  = type { i8 }
```

The first byte (i8) in each type is a dedicated tag, used to discriminate between the data constructors at runtime. Also visible in this example, the primary type `List` is created, which contains an array of 23 bytes, in addition to the discriminator byte. The length of this array of bytes is determined by the largest data constructor of the data type, with an addition of seven bytes to accommodate memory padding.

```
define %ListInt @Cons(ptr %cxt, i64 %arg_0, %List %arg_1) {
    %1 = alloca %List
    %2 = getelementptr %List, %List* %1, i64 0, i32 0
    store i8 0, i8* %2
    %3 = bitcast %List* %1 to %Cons*
    %4 = getelementptr %Cons, %Cons* %3, i64 0, i32 1
    store i64 %arg_0, ptr %4
    %5 = getelementptr %Cons, %Cons* %3, i64 0, i32 2
    %6 = call ptr @malloc(i64 24)
    store %List %arg_1, ptr %6
    store %List* %6, ptr %5
    %7 = load %List, ptr %1
    ret %List %7
}
```

Figure 3.13: *LLVM IR code for the constructor Cons*

The array is created to allow storing the values created by the data constructors on the stack, as a function always has to return a value of a fixed size. Using the `List` type, values created by both constructors can be stored on the stack, which will be coerced to the `List` type. In this example; the largest data constructor is `Cons`, which holds an integer of 8 bytes, and a pointer to the next element which is also 8 bytes, resulting in `List` holding an array of 23 bytes in total.

Any value of a recursive data constructor needs to be implemented using heap allocation which is the case for `Cons`. Without this, the size of the type would theoretically be infinite, as it would need to allocate enough memory on the stack to fit all possible values of the type. When implemented using heap allocation, constructors holds a pointer to the recursive value, instead of the actual value. To make this implementation easier, all occurrences of data types can be naively heap allocated. While only heap allocating recursive data types would lead to more efficient code, for simplicity, the current implementation heap allocates all data types held by other data types.

A function must also be created for each constructor. The function first needs to allocate enough memory for the primary type, then store the discriminator byte at the first spot in memory. Since `Cons` is the first constructor, it has a value of zero. The corresponding function must have a matching signature to the constructor it represents. The LLVM IR code in figure 3.13 describes the representation of the data constructor `Cons`.

3.8.3 Translating pattern matching

To represent pattern matching, branching at runtime is required. LLVM IR's `br` jumps are simple in nature and flexible. Either they are unconditional and always jump to the specified branch (`br label %Label`) or with a boolean value deciding which branch will be jumped to (`br i1 %cond, label %TrueLabel, label %FalseLabel`).

This example program pattern matches on the value `Cons 10 xs`:

```
case Cons 10 xs of
    Cons a xs => a + 1
    Nil       => 0
```

To translate a case expression, each pattern has to be broken down into multiple steps. If the value that is being scrutinized is a literal, a simple comparison and a branch on that result is enough, similar to how a `if` check would be implemented for an imperative language. If the check fails; jump to the next comparison and try again until one matches, or there are no more comparisons, and if so, crash or return an error. However, if the value is a data type, the generated comparisons have to be a bit more in-depth, almost like a chain of `if` statements. First, check the discriminator byte, and then repeat the process for all arguments. If an argument is a variable, then extract it from the data type and continue to the next argument. If an argument is a wildcard (`_`) then ignore it and proceed to the next argument. When a pattern has been matched successfully, evaluate its expression, and return the resulting value. The LLVM IR code in figure 3.14 represents how the aforementioned term could be translated.

```
    ; creation of the value
    %1 = call %List @Nil(ptr null)
    %2 = call %List @Cons(ptr null, i64 10, %List %1)
    ; the start of the pattern match
    %3 = alloca i64
    %4 = extractvalue %List %2, 0
    %5 = icmp eq i8 %4, 0
    br i1 %5, label %cons_success, label %nil_checks
cons_success:
    %6 = alloca %List
    %7 = load %Cons, ptr %6
    %a = extractvalue %Cons %7, 1
    %8 = extractvalue %Cons %7, 2
    %xs = load %List, ptr %8
    %9 = add i64 %$30a, 1
    store i64 %9, ptr %3
    br label %success
nil_checks:
    %10 = extractvalue %List %2, 0
    %11 = icmp eq i8 %10, 1
    br i1 %11, label %nil_success, label %failed_all_matches
nil_success:
    store i64 0, ptr %3
    br label %success
failed_all_matches:
    call void @crash()
success:
    %result = load i64, ptr %3
```

Figure 3.14: *LLVM IR code for pattern match described in section 3.8.3*

4 Runtime Environment

Functional programming languages tend to allocate more memory than imperative languages, as objects are copied for immutability (Aho et al., 2007). For example, Haskell can generate up to one gigabyte of garbage per second. As explicit memory management in other languages is error-prone, garbage collection is especially important in functional programming languages (Jones et al., 2012).

The language to implement Churf's garbage collector was chosen to be C++. In higher-level languages, abstractions are introduced for programs to be developed as type-, and memory-safe. For the garbage collector of Churf it was an important feature to have full control over memory, hence a lower-level language with fewer abstractions was chosen. Due to the chosen environment differing from the rest of the compiler, the garbage collector was designed as a library to be linked with the compiled Churf program, as depicted in figure 4.1.

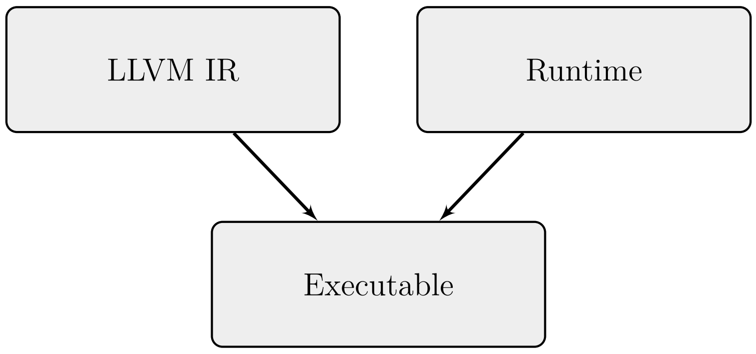


Figure 4.1: *The LLVM IR code from the compiler is linked with a static library of the runtime library to generate the final executable.*

4.1 Churf runtime functionality

Churf's runtime library offers three features: dynamic memory allocation, garbage collection of allocated memory, and a debugging tool for analyzing allocation and collection history throughout the process execution.

The library exposes the C++ function `Heap::alloc()`, a static member function of the `Heap` class, which is used for dynamically allocating memory. It accepts a non-negative integer representing the number of bytes to allocate. The function `Heap::alloc()` returns a void pointer, and was designed to behave similarly to the popular memory allocation function `malloc()` in the C standard library. The void pointer returned by `Heap::alloc()` points to some address and its value can be interpreted as a pointer of an appropriate type. The following C++ code provides an example of allocating memory using Churf's runtime library.

```
struct Point {
    int x, y, z;
};
Point *p = static_cast<Point *>(GC::Heap::alloc(sizeof(Point)));
```

Garbage collection is not exposed publicly in the runtime library, but acts as another step in the allocation process. The collector is triggered when an allocation request cannot be met and there is not enough heap memory available. A collection is then triggered and the allocation is retried after the collector is finished. If the retried allocation succeeds, it returns a void pointer as previously mentioned, otherwise, it throws an error since the heap ran out of memory. The runtime does not handle errors and if they heap is exhausted the cause of action is to crash the program.

A debugging tool was also developed for analyzing the behavior of memory operations. This tool is Churf's runtime profiler which tracks actions and timestamps of events related to the heap memory. If enabled, it records events throughout the execution of a Churf program and dumps them to a log file upon process termination. It is disabled by default but can be enabled by calling the function `set_profiler` as in the following example.

```
GC::Heap &heap = GC::Heap::the();
heap.set_profiler(true);
```

To use the library in LLVM IR, a Foreign Function Interface (FFI) is provided. The library's FFI was developed to assist the runtime library's integration with the compiler pipeline. When the library is compiled, functions defined under namespaces or in classes are renamed to machine-friendly conventions which are difficult to resolve in LLVM IR. The FFI provides top-level functions which are not renamed and can easily be used in LLVM IR. The following code is an example from the FFI, using the function `cheap_alloc` which is the global interface for `GC::Heap::alloc`, in LLVM IR.

```
%6 = call ptr @cheap_alloc(i64 24)
```

4.2 Library structure

The library consists of three classes defined under the namespace GC, and the FFI. The `heap` class provides most of the functionality of the library, such as allocation and garbage collection, while the other classes serve as building blocks for the `heap` class. The basic element of memory management is a chunk of memory, represented as a C++ structure.

```
namespace GC {
    struct Chunk {...};
    class GCEvent {...};
    class Heap {...};
    class Profiler {...};
}
void *cheap_alloc(unsigned long);
```

The `Chunk` structure models allocated memory and this structure is used extensively in the garbage collection algorithm to determine address spaces and reachable memory. A `Chunk` contains three key pieces of data; a boolean flag used by the collector to mark chunks and memory as reachable during collection, a constant pointer to the start address of the allocated memory, and a non-negative integer representing the number of bytes that are allocated starting from the start address. When an allocation request is made, a `Chunk` structure is created and stored in the `Heap` class, while its start address is returned as a pointer. The following code is an excerpt of the function `GC::Heap::alloc` which uses the structure for allocation.

```
Chunk *new_chunk = new Chunk(size,
    (uintptr_t *)(heap.m_heap_top)
);
...
heap.m_allocated_chunks.push(new_chunk);
return new_chunk->start;
```

`Chunk` structures are used extensively in the `Heap` class but have no purpose in a program using the library. The `Heap` class' main purpose is to expose a function used to allocate memory. The implementation follows the Singleton design pattern, allowing only one instance of the `Heap` class at any given time. In order for allocation and garbage collection to take place as intended, the instance has to be initialized. At the start of any Churf program, a call is made to the function `GC::Heap::init` which initializes the singleton instance. This function also retrieves information about the current stack frame, which is used for garbage collection.

Before program termination, the function `GC::Heap::dispose` should be called to destroy the singleton instance correctly, which simultaneously creates a log file if the profiler is enabled. Below is an example of a typical C++ program using the Churf runtime library.

```
int main()
{
    GC::Heap::init();
    int *p = static_cast<int *>(GC::Heap::alloc(sizeof(int)));
    GC::Heap::dispose();
    return 0;
}
```

Similarly to the `Chunk` structure, the building block of the profiler class is an event, which is modeled with the `GCEvent` class. This class has two key member variables, an enum signifying what the event corresponds to and, a timestamp of the event.

Types can signify the start of a specific function, such as when an allocation request is made or when a collection cycle starts, but can also represent operations on `Chunk` structures.

Some `GCEvent` instances also communicate extra information, such as when the member variable `m_type` has the value of `NewChunk`. The function `GC::Heap::alloc` creates new `Chunk` structures as needed, and the information these structures contain can be useful for debugging. Therefore, the `GCEvent` class holds member variables to provide some event types with additional information, as a pointer to a `Chunk` structure. To record an event, a `GCEventType` value is passed to the profiler together with additional information if needed.

```
auto new_chunk = new Chunk(size,
    (uintptr_t *)(heap.m_heap + heap.m_size)
);
if (profiler_enabled)
        Profiler::record(NewChunk, new_chunk);
```

When the profiler class receives an event type, it creates an instance of the `GCEvent` class and stores it in a list of events, sorted by the timestamps of the events. If the profiler is enabled, it performs very little work during execution. It is at program termination that the profiler writes all the events to a log file. This process is triggered when the `Heap` class is disposed of.

There are three modes by which the profiler can log information. By default, the profiler logs every event separately and covers all of the available types. This produced log files too large to fully comprehend and introduced the development of another mode. The profiler can filter events by function types such as `AllocStart`. When filtering events on function types, the profiler summarizes the flow of the program by counting how many events of the same type occurred in a sequence. To specify the type of log file, a function call can be made to the function `GC::Heap::set_profiler_log_options`.

```
GC::Heap &heap = GC::Heap::the();
heap.set_profiler_log_options(GC::FunctionCalls);
```

4.3 Allocation strategy

The processes of allocating and reclaiming memory are closely associated with each other in garbage collectors and the design choices of either impose limitations on the other (Jones et al., 2012). The allocation function in Churf's runtime library implements a first-fit sequential allocation algorithm. The paradigm of sequential algorithms was initially chosen because it is a simple strategy and easy to implement (Jones et al., 2012). Because of its simplicity, the process of allocating memory is relatively short and fast. It is for the same reason, however, that the address space that is allocated might not be the best fit for the allocation, resulting in worse overall performance. Though, the speed of the allocation process was considered more important than overall program performance, as functional languages tend to produce a lot of allocations.

In first-fit allocation, the allocator looks for the first available memory space large enough to satisfy the allocation request and splits it if it is larger than the specified size. In the Churf runtime library, this is implemented with a free list, containing all previously allocated and collected chunks of memory. As long as no chunks have been freed during garbage collection, `GC::Heap::alloc` allocates heap memory sequentially.

A collection is triggered when the top of the heap is too small to satisfy the allocation request, which means that the most recent successful allocation request will point to a chunk of memory close to the top of the heap. After a collection is triggered, the free list is populated with chunk structures reclaimed by the garbage collector. These freed chunks commonly come from the bottom of the heap, as they are the oldest or first allocations made.

When the free list is not empty, first-fit allocation is introduced as the primary allocation algorithm. Since the top of the heap contains the most recent allocations, they are more likely to still be reachable. Therefore, the allocator looks to allocate at the start of the heap and looks for the first available chunk of memory that will suffice. This also helps to reduce the amount of memory fragmentation.

4.4 The choice of collection algorithm

The choice of the Mark-Sweep algorithm came down to two factors; complexity and independence. Complexity in the sense of how difficult the algorithm is to implement. Both in itself but also relative to the behavior of the language. Mark-Compact is an example of a more complex algorithm that could in most cases be considered better than Mark-Sweep. It is built on the same structure as Mark-Sweep but additionally compacts the heap to reduce fragmentation.

Another factor was independence and how well the collector could be implemented in parallel with the compiler. As mentioned in 2.3.3, Mark-Sweep is a *stop-the-world* algorithm. The minimal version of it enforces no constraints on the compiler (Jones et al., 2012). Therefore, it is a great choice for implementing independently of the compiler.

Another design choice was to implement a conservative garbage collector, as opposed to an accurate one. The distinction between accurate and conservative collectors stems from how they identify objects on the heap from the stack. An accurate collector receives information about the stack from the compiler, decided at compile-time. Mainly, which memory addresses contain pointers to allocated objects (Jones et al., 2012). This ensures that all live objects on the heap are identified. An accurate collector can only exist if the language it operates on is type-safe (Aho et al., 2007). Churf is type-safe, thus implementing an accurate collector is possible.

On the other hand, a conservative collector scans the stack for pointers without any previous knowledge. In order to not neglect any live objects, everything that resembles a pointer is assumed to be one (Jones et al., 2012). However, this leads to the possibility that unrelated data might be scanned, making it less effective than an accurate collector. Although, it does not impose any overhead on the compiler since it requires no information from it. Given that a primary aim was to make the development of the collector as independent as possible, a conservative collector was chosen over an accurate one since it does not have to communicate with the compiler.

4.5 Mark-Sweep implementation

Churf's garbage collector was designed to only perform a collection if it is needed. The collector is only invoked when an allocation fails due to the memory managed either being full or too fragmented. The implementation is an extended version of the Mark-Sweep algorithm presented in section 2.3.3. It divides the sweep-phase into sweeping and freeing. The `free` function in the collector does not simply remove the memory address of the given chunk, it is more involved in order to combat fragmentation of the heap. The process will be elaborated on in this chapter.

In order to locate all the live objects in the running program, all stack addresses need to be scanned. When the collector is initialized, the top of the stack is stored in the member variable `m_stack_top`. Then, when the collection phase start, the bottom of the stack is captured which is the frame address of the current function `collect`. All addresses in between the top and the bottom of the stack are stored in the vector `roots`, by the function `find_roots`. Additionally, to have fast lookup of allocated chunks in the mark-phase, a hash table of all the allocated chunks is created and stored in `m_chunk_table`. The flow of the collection phase is as is shown in 4.2.

```
void Heap::collect()
{
    Heap &heap = Heap::the();
    stack_bottom =
        reinterpret_cast<uintptr_t *>(__builtin_frame_address(0));
    vector<uintptr_t *> roots;
    find_roots(stack_bottom, roots);
    m_chunk_table = create_table();
    mark(roots);
    sweep(heap);
    free(heap);
}
```

Figure 4.2: *The collection function in the runtime library.*

The mark phase iterates over all the addresses in `roots`. For every address, it calls a helper function `find_chunks`, displayed in figure 4.3. `find_chunks` determines if the address points to some allocated memory, by performing a lookup on `m_chunk_table`. If it does and the chunk is not already marked, it is marked, to indicate that it is used by the program. Furthermore, each address space of a marked chunk is stored in a queue, `chunk_spaces`. That is, a pair consisting of the start-, and end-address of the chunk. This step is crucial to locate any reachable chunks in the program that are not directly reachable from the stack.

```
void Heap::find_chunks(
    uintptr_t *stack_addr,
    queue<pair<uintptr_t, uintptr_t>> &chunk_spaces)
{
    Heap &heap = Heap::the();

    auto it = heap.m_chunk_table.find(*stack_addr);
    if (it != heap.m_chunk_table.end())
    {
        auto chunk = it->second;

        if (!chunk->m_marked)
        {
            auto c_start = reinterpret_cast<uintptr_t>(chunk->m_start);
            auto c_size = reinterpret_cast<uintptr_t>(chunk->m_size);
            auto c_end = reinterpret_cast<uintptr_t>(c_start + c_size);

            chunk->m_marked = true;
            chunk_spaces.push(std::make_pair(c_start, c_end));
        }
    }
}
```

Figure 4.3: *The helper function used in* `mark`

Consider a linked list of 1000 elements. The memory representing this list, in the runtime library, consists of 1000 chunks, each one holding a pointer to the next one, except the last. Only the head of the list is visible from the stack. To locate every chunk and mark them as reachable, it is necessary to further go down the chain of chunks until the end. This is achieved by continuously adding chunk spaces until none is found, by calling `find_chunks`. The function `mark`, shown in figure 4.4, starts with locating chunks on the stack, then continues searching for chunks until `chunk_spaces` is empty, which indicates that every reachable chunk is scanned and marked.

Following from the mark-phase is the sweeping. The process is simple, `sweep` iterates through `m_allocated_chunks`. All marked chunks have their mark-flag reset for the next collection cycle. The chunks which are not marked are placed into a vector `m_freed_chunks`, to either be recycled or deleted, in the freeing-phase. Then they are removed from `m_allocated_chunks`, since they are no longer in use by the program.

Lastly, the `free` function is called. Due to the nature of the Mark-Sweep algorithm and its tendency to fragment the heap, this function is rather complex, since it tries to combat memory fragmentation, without moving pointers. The idea is to reuse old chunks, that is, chunks that have been allocated but are no longer in use, as

```
void Heap::mark(vector<uintptr_t *> &roots)
{
    auto iter = roots.begin(), end = roots.end();
    queue<pair<uintptr_t, uintptr_t>> chunk_spaces;

    while (iter != end)
    {
        find_chunks(*iter++, chunk_spaces);
    }
    while (!chunk_spaces.empty())
    {
        auto range = chunk_spaces.front();
        chunk_spaces.pop();

        auto addr_bottom = reinterpret_cast<uintptr_t *>(range.first);
        auto addr_top = reinterpret_cast<uintptr_t *>(range.second);

        while (addr_bottom < addr_top)
        {
            find_chunks(addr_bottom, chunk_spaces);
            addr_bottom++;
        }
    }
}
```

Figure 4.4: *The function that marks all reachable memory chunks, from the stack.*

much as possible, as elaborated on in chapter 4.3. To realize this, `free` only deletes chunks if a certain threshold is reached. `FREE_THRESH` is a heuristic variable and is relative to the size of the heap.

```
if (heap.m_freed_chunks.size() > FREE_THRESH)
{
    while (heap.m_freed_chunks.size()
    {
        auto chunk = heap.m_freed_chunks.back();
        heap.m_freed_chunks.pop_back();
        heap.m_size -= chunk->m_size;
        delete chunk;
    }
}
```

If the amount of freed chunks are below the threshold, another freeing-function is used, `free_overlap`. The implementation of `free_overlap` is present in C.2.

5
Demonstration

The Churf compiler can be used by calling the compiler in a command line environment and supplying the intended `.crf` file to be compiled. The resulting executable will be stored in a folder called `output`, located where the compiler is called from. The executable has no dependencies besides `libc` and should work on most Linux distributions. Following are the command-line flags that can be used when compiling:

Flag	Verbose	Description
`-d`	`--debug`	Print debug messages. Implies `--log-intermediate`
`-t`	`--type-checker <bi/hm>`	Choose type checker. Possible options are Bidirectional(`bi`) and Hindley-Milner(`hm`)
`-m`	`--disable-gc`	Disables the garbage collector and uses `malloc` instead
`-l`	`--log-intermediate`	Log intermediate languages
`-p`	`--disable-prelude`	Do not include the prelude
	`--help`	Print a message containing this

5.1 Demonstration of Churf

This section demonstrates Churf by showcasing the Quicksort algorithm in figure 5.1, the skew heap data structure in figure 5.2, and finally a call-by-name lambda calculus interpreter in figure 5.3. The definition of `List`, addition, and subtraction are omitted as they are included in the prelude of Churf.

```
filter : forall a. (a -> Bool) -> List a -> List a
filter p xs = case xs of
    Nil       => Nil
    Cons x xs => case p x of
        True  => Cons x (filter p xs)
        False => filter p xs

.++ : forall a. List a -> List a -> List a
.++ as bs = case as of
    Nil       => bs
    Cons x xs => Cons x (xs ++ bs)

quicksort : List Int -> List Int
quicksort xs = case xs of
    Nil       => Nil
    Cons a as => let smaller = quicksort (filter (\y. y < a) xs) in
                 let bigger  = quicksort (filter (\y. a < y) xs) in
                 smaller ++ (Cons a bigger)
```

Figure 5.1: *The Quicksort algorithm implemented in Churf.*

5.2 Garbage collector

The runtime library provides support for memory allocation of recursive data types in Churf. The runtime library provides an interface for LLVM IR code to allocate memory on the heap, and a garbage collection algorithm to reclaim unused memory. The library allocates heap memory with the first-fit allocation strategy and collects unused memory with an implementation of the Mark-Sweep algorithm.

The difficulty of reclaiming objects lies in determining what memory is reachable or used. Allocated objects may point to other objects and where this pointer resides in the allocated object is undetermined. Linked lists are an example of this as they contain pointers to the next element in the list. The code in figure 5.4 is a Churf program that allocates 1000 linked lists, each of 10000 elements.

In figure 5.4, each recursive call to the function `revRange` creates a node that is heap allocated. The head of the list is stack-allocated in LLVM IR and is popped off the stack when the `revRange` function exits recursion. Depending on the set size of the heap, one allocation will fail and trigger a collection. Since the head of the old list is no longer on the stack, the garbage collector will not mark it as reachable and then reclaim all that lists' memory.

The memory usage of the program depicted in figure 5.4 with different parameters is shown in table 5.1. Four different versions of the program are tested. The first program allocates a single list with 10000 elements, and the second allocates 1000 lists with an equal amount of elements. Both of these programs are tested with and without the garbage collector, proving the usefulness of a garbage collector. For all

```
data Skewheap where
    Empty : Skewheap
    Node  : Skewheap -> Int -> Skewheap -> Skewheap

data Maybe a where
    Nothing : Maybe a
    Just    : a -> Maybe a

peek : Skewheap -> Maybe Int
peek tree = case tree of
    Node l x r => Just x
    _ => Nothing

pop tree = case tree of
    Node l x r => Just (Pair x (merge l r))
    Empty => Nothing

merge tree1 tree2 = case tree1 of
    Node left1 val1 right1 => case tree2 of
        Node left2 val2 right2 => case val1 < val2 of
            True => Node
                        (merge right1 (Node left2 val2 right2))
                        val1
                        left1
            False => Node
                        (merge right2 (Node left1 val1 right1))
                        val2
                        left2
        _ => tree1
    Empty => tree2


singleton x = Node Empty x Empty

insert x tree = merge (singleton x) tree
```

Figure 5.2: *The Skew heap data structure implemented in Churf.*

```
data Exp where
    EVar : Char -> Exp
    EAbs : Char -> Exp -> Exp
    EApp : Exp -> Exp -> Exp
    EInt : Int -> Exp
    EAdd : Exp -> Exp -> Exp

data Val where
    VInt     : Int -> Val
    VClosure : Cxt -> Char -> Exp -> Val

data Cxt where
    Cxt : List (Pair Char Val) -> Cxt

lookup : Char -> Cxt -> Val
lookup x cxt = case cxt of
    Cxt ps => case ps of
        Cons p ps => case p of
            Pair y v => case (asciiCode x) == (asciiCode y) of
                True  => v
                False => lookup x (Cxt ps)

insert : Char -> Val -> Cxt -> Cxt
insert x v cxt = case cxt of { Cxt ps => Cxt (Cons (Pair x v) ps) }

eval : Cxt -> Exp -> Val
eval cxt exp = case exp of
    EVar x => case lookup x cxt of
        VInt i             => VInt i
        VClosure delta x e => eval delta e
    EAbs x e => VClosure cxt x e
    EApp e1 e2 => case eval cxt e1 of
        VClosure delta x f =>
            let v = VClosure cxt x e2 in
            eval (insert x v delta) f
    EInt i => VInt i
    EAdd e1 e2 =>
        let i1 = case eval cxt e1 of { VInt i => i } in
        let i2 = case eval cxt e2 of { VInt i => i } in
        VInt (i1 + i2)

-- (λx.x) (λx.x + 100) 200
exp = EApp (EAbs 'x' (EVar 'x'))
           (EApp (EAbs 'x' (EAdd (EVar 'x') (EInt 100))) (EInt 200))

main = case eval (Cxt Nil) exp of { VInt i => i }
```

Figure 5.3: *A call-by-name lambda calculus interpreter implemented in Churf.*

```
main : Int
main = allocate 0 1000

allocate : forall a. a -> Int -> Int
allocate x n = case n of
    0 => 0
    n => allocate (revRange 10000) (n - 1)

data List a where
    Nil : List a
    Cons : a -> List a -> List a

revRange : Int -> List Int
revRange x = case x of
    0 => Cons x Nil
    x => Cons x (revRange (x - 1))
```

Figure 5.4: *Allocating linked lists in Churf as a benchmark for garbage collection.*

versions, the maximum heap size was set to 314 kilobytes, which is roughly 130% of the size of one list in the program. Each node uses 24 bytes, which means that the total heap size of all nodes is 240 kilobytes.

In programs run without the garbage collector, calls to allocate memory through the runtime library are swapped with calls to the function `malloc` in the C standard library. In the programs allocating only a single list, the amount of unfreed memory remains the same. This is due to internal variables in the runtime library not being disposed of correctly at program exit and is a bug in the library. Optimally, the amount of unfreed memory at program exit should be none.

The two programs allocating a 1000 lists have significantly higher memory usage. Without the garbage collector, none of the calls to `malloc` will be freed or collected, and there is a great amount of unfreed memory. However, with the garbage collector all but ~176 kB of the total ~481 MB is collected, indicating that the of Churf's garbage collector works.

GC	**Number of lists**	**Memory usage**	**Unfreed memory**
Without GC	1	~314 kB	~240 kB
Without GC	1000	~240 MB	~240 MB
With GC	1	~891 kB	~240 kB
With GC	1000	~481 MB	~176 kB

Table 5.1: *Memory analysis from running the benchmark program depicted in figure 5.4 with and without the garbage collector.*

6 Discussion

This chapter discusses the differences between System F and Churf, the advantages of using LLVM IR as the target language, and the choice of garbage collection algorithm.

6.1 Differences between System F and Churf

Syntactically Churf does not adhere to System F as specified by the rules in figure 2.1. Type abstraction ($\Lambda\alpha.e$) and type application ($e[A]$) are not explicitly written in Churf. Instead, type abstraction is assumed on terms with polymorphic types, and type application is inferred during the term application of a polymorphic function. To illustrate this difference, consider the following identity function in System F and Churf, respectively.

$\Lambda\alpha\,.\,\lambda x\,.\,x$

```
identity = \x . x : forall a. a -> a
```

And here is how type application differs syntactically between System F and Churf.

$(\Lambda\alpha\,.\,\lambda x\,.\,x)[\beta] : \beta \to \beta$

```
identity_int = identity : Int -> Int
```

Another desirable property that is missing is inference of polymorphic instantiation (impredicativity). As a result, some terms require seemingly unnecessary type annotations, for example `Cons id Nil`. This also means that polymorphic types are second-class in the type system. To use impredicative types in Churf, `Cons` needs the following type annotation.

```
(Cons : (forall a. a -> a)
      -> List (forall a. a -> a)
      -> List (forall a. a -> a)) id Nil
```

The introduction of data types and pattern matching in addition to System F enables a simple syntax for expressing data structures, and the deconstruction of them. Although data types and pattern matching could be expressed using Böhm-Berarducci encoding (Böhm & Berarducci, 1985), it ends up being less convenient than having them implemented in the language. The translation of data types to the corresponding LLVM IR representation is also very similar.

6.2 LLVM IR as a target language for a functional language

The intermediate transformations of Churf result in a language not derived from lambda calculus, but an expression-based language in SSA form. As LLVM IR is in SSA form, the translation from the intermediate language to LLVM IR is straightforward. The more interesting feature of LLVM IR is the type system. By having a type system in a lower-level language, ill-typed code generation is prevented at compile time. Most notably, the compiler makes heavy use of LLVM IR's `getelementptr` to perform pointer arithmetic using type information. In a target language without types and sizes known statically this would have to be done manually.

LLVM provides a tool to perform tail-call optimization, among other optimizations, called `opt`. The tool performs various optimizations and generates new LLVM IR code. A caveat is that the generated LLVM IR code has to be written in a manner that can be tail-call optimized, as there is no guarantee that the tool will perform tail-call optimization otherwise. Churf utilizes this tool to enable tail-call recursion.

6.3 Alternative collection algorithm

The runtime library works well with the compiler and Churf programs, but suffers in performance from algorithm complexity. Although the goal of the library in the project is met, there is still potential for improvement. The time complexity of the allocation and collection strategies can be optimized, by the use of different data structures and alternative implementation methods. Another way of improving performance is the implementation of a different garbage collection algorithm that better suits Churf programs.

There are several other algorithms for garbage collection. Mark-Compact is an algorithm derived to some degree from Mark-Sweep and aims to minimize the memory fragmentation that Mark-Sweep suffers from. Mark-Compact collectors are designed with special support in mind, which can either be memory analysis from the compiler or the type system of the language (Jones et al., 2012). In turn, this can guarantee the ability to distinguish data from pointers and to compact previously allocated memory for efficiency.

If independence between the compiler and runtime was not a priority and the use case of the language was broader, Mark-Compact could have been implemented. Since Mark-Compact makes allocation more efficient and makes better use of the heap memory over a running program's lifetime. It also eliminates the possibility of heap fragmentation which is preferred in larger programs (Jones et al., 2012).

7
Conclusion

The goal of this project was to implement a compiler for a functional programming language as well as a runtime library for the language. The compiler covers syntactic analysis, semantic analysis, intermediate representation, and code generation. In addition to the compiler, a runtime library has been developed to support heap-allocated memory. The goals set in 1.2 have all been met, but there is room for improvement.

The Churf compiler is capable of compiling programs that adhere to the language specification, producing LLVM IR code that can then be compiled to executable machine code. In addition to System F Churf introduces data types and pattern matching. As shown in figures 5.1 and 5.4, it is possible to write programs performing common tasks such as creating and sorting lists. But it is also possible to create larger programs serving more specialized needs, for example the lambda calculus interpreter depicted in figure 5.3.

Opting for different design choices or a more intimate development process between the different parts of the compiler of the project could have resulted in a smarter compiler or a better garbage collector. Given more development time on the code generator, a larger set of Churf programs could be translated to tail-call optimizable code. If the compiler could perform memory analysis the garbage collector would have been accurate and more performant. Regardless, the project was a success and achieved the goals set at the start of the project.

Bibliography

Abel, A., et al. (2023). *The BNF converter* (Version 2.9.4.1) [Computer software]. Chalmers University of Technology; University of Gothenburg. https://bnfc.digitalgrammars.com/

Abel, A., Coquand, T., & Dybjer, P. (2008). Verifying a semantic $\beta\eta$-conversion test for martin-löf type theory. In P. Audebaud & C. Paulin-Mohring (Eds.), *Mathematics of program construction* (pp. 29–56). Springer Berlin Heidelberg.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, techniques, & tools* (2nd ed). Pearson/Addison Wesley.

Böhm, C., & Berarducci, A. (1985). Automatic synthesis of typed Λ-programs on term algebras [Third Conference on Foundations of Software Technology and Theoretical Computer Science]. *Theoretical Computer Science*, *39*, 135–154. https://doi.org/https://doi.org/10.1016/0304-3975(85)90135-5

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics*, *33*(2), 346–366. Retrieved March 17, 2023, from http://www.jstor.org/stable/1968337

Church, A. (1940). A formulation of the simple theory of types. *The Journal of Symbolic Logic*, *5*(2), 56–68. https://doi.org/10.2307/2266170

Cooper, K., & Torczon, L. (2003). *Engineering a compiler* (1st ed.). Morgan Kaufmann.

Curry, H. B. (1934). Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, *20*(11), 584–590. Retrieved April 25, 2023, from http://www.jstor.org/stable/86796

Curry, H., & Feys, R. (1958). *Combinatory logic*. North-Holland Publishing Company. https://books.google.se/books?id=fEnuAAAAMAAJ

Dunfield, J., & Krishnaswami, N. (2021). Bidirectional typing. *ACM Comput. Surv.*, *54*(5). https://doi.org/10.1145/3450952

Dunfield, J., & Krishnaswami, N. R. (2013). Complete and easy bidirectional typechecking for higher-rank polymorphism. *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 429–442. https://doi.org/10.1145/2500365.2500582

Dunfield, J., & Pfenning, F. (2004). Tridirectional typechecking. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 281–292. https://doi.org/10.1145/964001.964025

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur* (Thèse d'État). Université Paris VII.

Girard, J.-Y. (1986). The system F of variable types, fifteen years later. *Theoretical computer science*, *45*, 159–192.

Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, *146*, 29–60. Retrieved April 11, 2023, from http://www.jstor.org/stable/1995158

Howard, W. A. (1980). The formulae-as-types notion of construction. In H. Curry, H. B., S. J. Roger, & P. Jonathan (Eds.), *To h. b. curry: Essays on combinatory logic, lambda calculus, and formalism*. Academic Press.

Johnsson, T. (1985). Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud (Ed.), *Functional programming languages and computer architecture* (pp. 190–203). Springer Berlin Heidelberg.

Jones, R., Hosking, A., & Moss, E. (2012). *The garbage collection handbook: The art of automatic memory management*. Chapman Hall/CRC.

Lattner, C. (2002). *LLVM: An Infrastructure for Multi-Stage Optimization* (Master's thesis) [*See* `http://llvm.cs.uiuc.edu`.]. Computer Science Dept., University of Illinois at Urbana-Champaign. Urbana, IL.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, *3*(4), 184–195. https://doi.org/10.1145/367177.367199

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, *17*(3), 348–375.

Norell, U. (2007). Towards a practical programming language based on dependent type theory.

Peyton Jones, S., & Lester, D. (1992). *Implementing functional languages: A tutorial*. Prentice Hall. https://www.microsoft.com/en-us/research/publication/implementing-functional-languages-a-tutorial/

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Shields, M. (2007). Practical type inference for arbitrary-rank types. *J. Funct. Program.*, *17*(1), 1–82. https://doi.org/10.1017/S0956796806006034

Pierce, B. C., & Turner, D. N. (2000). Local type inference. *ACM Trans. Program. Lang. Syst.*, *22*(1), 1–44. https://doi.org/10.1145/345099.345100

Ranta, A. (2012). *Implementing Programming Languages*. College Publications.

LLVM Developer Group. (2023a). Getting Started with the LLVM System. Retrieved April 8, 2023, from https://llvm.org/docs/GettingStarted.html#hardware

LLVM Developer Group. (2023b). Getting Started with the LLVM System. Retrieved April 8, 2023, from https://llvm.org/docs/FAQ.html

LLVM Developer Group. (2023c). The LLVM Compiler Infrastructure Project. Retrieved April 8, 2023, from https://llvm.org/

LLVM Developer Group. (2023d). Llvm Language Reference Manual. Retrieved April 8, 2023, from https://llvm.org/docs/LangRef.html

Reynolds, J. C. (1974). Towards a theory of type structure. *Programming Symposium, Proceedings Colloque Sur La Programmation*, 408–423.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, *12*(1), 23–41. https://doi.org/10.1145/321250.321253

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts*. John Wiley & Sons, Inc.

Streib, J. T. (2020). Guide to assembly language: A concise introduction. *Guide to Assembly Language*.

Wadler, P. (1990). Comprehending monads. *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 61–78. https://doi.org/10.1145/91556.91592

Wells, J. (1999). Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, *98*(1), 111–156. https://doi.org/https://doi.org/10.1016/S0168-0072(98)00047-5

A
Bidirectional type system rules

$\boxed{\Gamma \vdash e \uparrow A \dashv \Delta}$

In context Γ, e checks against type A with output context Δ

$$\frac{\Gamma, \alpha \vdash e \uparrow A \dashv \Delta, \alpha, \Theta}{\Gamma \vdash e \uparrow \forall \alpha . A \dashv \Delta}\ \forall\text{I} \qquad \frac{\Gamma, (x : A) \vdash e \uparrow B \dashv \Delta, (x : A), \Theta}{\Gamma \vdash \lambda x . e \uparrow A \to B \dashv \Delta}\ \to\text{I}$$

$$\frac{\Gamma \vdash e \downarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \uparrow B \dashv \Delta}\ \text{Sub} \qquad \frac{\text{typeof}(lit) = A}{\Gamma \vdash lit \uparrow A \dashv \Gamma}\ \text{Lit}$$

$$\frac{\Gamma \vdash e \downarrow A \dashv \Theta \qquad \Theta \vdash \overline{p \Rightarrow e} :: [\Theta]A \uparrow [\Theta]B \dashv \Delta}{\Gamma \vdash \text{case } e \text{ of } \overline{p \Rightarrow e} \uparrow B \dashv \Delta}$$

$\boxed{\Gamma \vdash e \downarrow A \dashv \Delta}$

In context Γ, e infers output type A with output context Δ

$$\frac{}{\Gamma \vdash lit \downarrow \text{typeof}(lit) \dashv \Gamma}\ \text{Lit}\downarrow \qquad \frac{\Gamma \ni (x : A)}{\Gamma \vdash x \downarrow A \dashv \Gamma}\ \text{Var}$$

$$\frac{\Gamma \vdash rec(x)}{\Gamma \vdash x \downarrow \hat{\alpha} \dashv \Gamma, (x : \hat{\alpha})}\ \text{VarRec}$$

$$\frac{\Gamma \vdash \lambda \overline{y} . e \downarrow A \dashv \Theta_1 \qquad \Theta_1, (x : A) \vdash e \downarrow B \dashv \Delta, (x : A), \Theta_2}{\Gamma \vdash \text{let } x\, \overline{y} = e \text{ in } e' \downarrow B \dashv \Delta}\ \text{Let}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash e \uparrow A \dashv \Delta}{\Gamma \vdash (e : A) \downarrow A \dashv \Delta}\ \text{Anno} \qquad \frac{\Gamma, \hat{\alpha}, \hat{\beta}, (x : \hat{\alpha}) \vdash e \uparrow \hat{\beta} \dashv \Delta, (x : \hat{\alpha}), \Theta}{\Gamma \vdash \lambda x . e \downarrow \hat{\alpha} \to \hat{\beta} \dashv \Delta}\ \to\text{I}\downarrow$$

$$\frac{\Gamma \vdash e_1 \downarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \bullet e_2 \Downarrow C \dashv \Delta}{\Gamma \vdash e_1\ e_2 \downarrow C \dashv \Delta}\ \to\text{E} \qquad \frac{\Gamma \ni (K : A)}{\Gamma \vdash K \downarrow A \dashv \Gamma}\ \text{Inj}$$

$\boxed{\Gamma \vdash A \bullet e \Downarrow C \dashv \Delta}$

In context Γ, applying a function of type A to e infers output type C with output context Δ

$$\frac{\Gamma, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A \bullet e \Downarrow C \dashv \Delta}{\Gamma \vdash \forall \alpha . A \bullet e \Downarrow C \dashv \Delta}\ \forall\text{App} \qquad \frac{\Gamma \vdash e \uparrow A \dashv \Delta}{\Gamma \vdash A \to C \bullet e \Downarrow C \dashv \Delta}\ \to\text{App}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \alpha = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash e \uparrow \hat{\alpha}_1 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} \bullet e \Downarrow \hat{\alpha}_2 \dashv \Delta}\ \hat{\alpha}\text{App}$$

Figure A.1: *Typing rules of the bidirectional typing system that is presented in section 3.5. The system is based on the algorithmic typing system by Dunfield and Krishnaswami (2013).*

B

Memory layout

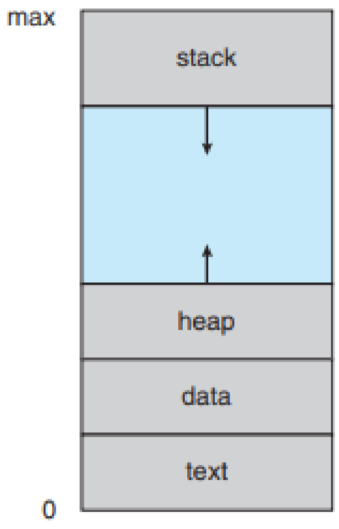


Figure B.1: *Memory layout of a program loaded in memory (Silberschatz et al., 2018).*

C

Runtime library

```
void *Heap::alloc(size_t size) {
    Heap &heap = Heap::the();
    Chunk *reused_chunk = heap.try_recycle_chunks(size);
    if (reused_chunk != nullptr)
        return static_cast<void *>(reused_chunk->m_start);
    if (heap.m_heap_start + heap.m_size > MAX_HEAP_SIZE) {
        heap.collect();
        if (heap.m_heap_start + heap.m_size > MAX_HEAP_SIZE) {
            throw std::runtime_error(
                std::string("Error:␣Heap␣out␣of␣memory")
            );
        }
    }
    Chunk *new_chunk = new Chunk(
        size,
        (uintptr_t *)(heap.m_heap_start + heap.m_size)
    );
    heap.m_size += size;
    return static_cast<void *>(new_chunk->m_start);
}
```

Figure C.1: *First-fit allocation implementation in the runtime library.*

```
void Heap::free_overlap(Heap &heap)
{
    std::vector<Chunk *> filtered;
    size_t i = 0;
    auto prev = heap.m_freed_chunks[i++];
    prev->m_marked = true;
    filtered.push_back(prev);
    for (; i < heap.m_freed_chunks.size(); i++)
    {
        prev = filtered.back();
        auto next = heap.m_freed_chunks[i];
        auto p_start = (uintptr_t)(prev->m_start);
        auto p_size = (uintptr_t)(prev->m_size);
        auto n_start = (uintptr_t)(next->m_start);
        if (n_start >= (p_start + p_size))
        {
            next->m_marked = true;
            filtered.push_back(next);
        }
    }
    heap.m_freed_chunks.swap(filtered);

    bool profiler_enabled = heap.m_profiler_enable;
    // After swap m_freed_chunks contains still available chunks
    // and filtered contains all the chunks, so delete unused chunks
    for (Chunk *chunk : filtered)
    {
        // if chunk was filtered away, delete it
        if (!chunk->m_marked)
        {
            if (profiler_enabled)
                Profiler::record(ChunkFreed, chunk);
            heap.m_size -= chunk->m_size;
            delete chunk;
        }
        else
        {
            chunk->m_marked = false;
        }
    }
}
```

Figure C.2: *Implementation of `free_overlap`, that deletes overlapping unused memory chunks.*