Dylan Moss

Automatic Parallelisation using Effect Tracking and Cost Analysis

Computer Science Tripos – Part II
Queens' College

January 29, 2024

Declaration

I, Dylan Moss of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: Dylan Moss
Date: January 29, 2024

Proforma

Candidate number:	2432D
Project Title:	**Automatic Parallelisation using** **Effect Tracking and Cost Analysis**
Examination:	**Computer Science Tripos – Part II, 2023**
Word Count:	11991[1]
Code Line Count:	6128[2]
Project Originator:	The Candidate and Prof. Alan Mycroft
Project Supervisor:	Prof. Alan Mycroft

Original Aims of the Project

The core aim was to implement automatic parallelisation into the compiler of a mid-featured, imperative programming language. This language, dubbed *Kautuka*, implements a sensible subset of Go. The automatic parallelisation compiler translates Kautuka into parallelised Go, utilising *side-effect tracking* and *cost analysis* to produce *safe* and *efficient* parallel code. This eliminates the need for parallelisation primitives (which require complex mental models) to exploit the benefits of multicore parallelism. Kautuka's performance would be evaluated against sequential Go. Extensions include: implementing an inference algorithm, adding I/O operations (with aliasing analysis), and developing tools which integrate Kautuka into existing workflows.

Work Completed

Exceeded all success criteria and the majority of extensions. Kautuka supports variables, control flow, functions, and user I/O operations. Kautuka produces parallelised code which outperforms sequential Go on I/O-bound programs (the target candidates for performance improvements). I extended side-effect tracking with aliasing analysis, to track file references in file-I/O operations. I implemented an inference algorithm into cost analysis, allowing me to translate of hundreds of lines of Go into Kautuka with minimal overhead. I demonstrated how Kautuka can be integrated into existing workflows, enabling its gradual adoption into established Go codebases.

Special Difficulties

None.

[1] This word count was computed using `texcount`.

[2] This code line count was computed using `cloc` (excluding autogenerated test output).

Contents

Chapter 1

Introduction

Since the mid 2000s, single-core processor performance has stagnated, with performance gains primarily achieved through *multicore parallelism*. Modern programming languages allow programmers to specify parallel execution behaviour (how code is executed across multiple cores) to improve program performance.

However, parallelisation is inherently non-deterministic, resulting in *race conditions*: where a program's output depends on the interleaving order of its threads. Race conditions give rise to a new category of bugs known for their elusive nature, only manifesting in certain thread interleavings. Since many programming languages do not ensure correctness in parallel programs (that is the absence of race conditions), the onus is on the programmer to detect these bugs, which is both challenging and prone to error. In addition, executing programs across multiple cores necessitates inter-core communication: a costly operation which may result in worse performance compared to the original sequential program. Despite its potential for enhancing performance, multicore parallelism poses significant challenges when writing *safe* and *efficient* parallel programs.

Automatic parallelisation compilers alleviate these problems by automatically compiling sequential code into parallel code, ensuring the produced code is safe and efficient with minimal programming overhead. There are two main approaches to automatic multicore parallelisation. The first is *data-level parallelisation*: running the same operation on multiple memory locations at once. This low-level approach is effective at parallelising loops and other vectorisation operations, and is often found in mainstream imperative programming language compilers. It is best suited at optimising data-intensive computations, where loops tend to make up most of the execution time [1]. The alternative approach is *task-based parallelisation*: parallelising groups of code representing an individual task. In this project, each code block is considered a task — denoted by wrapping a group of code in a pair of isolated curly braces. This naturally extends the typical use of code blocks: grouping together statements which perform the same "task" into a new scope, as threads (parallelised code blocks) follow the same scoping rules as code blocks. The pseudocode below illustrates an example of task-based parallelisation.

```
{
  for i in range(1000) {
    print(i)
  }
}

{
  x := 0
  for j in range(1000) {
    x += j
  }
}
```

$\xrightarrow{\text{parallelise}}$

```
new thread(){
  for i in range(1000) {
    print(i)
  }
}.run()

new thread(){
  x := 0
  for j in range(1000) {
    x += j
  }
}.run()
```

Task-based automatic parallelisation, also known as *high-level automatic parallelisation*, aims to improve the performance of programs containing large, independent tasks. Implementations in literature typically focus on toy functional languages as opposed to imperative languages, as this approach requires complex static analysis. However, despite the prevalence of imperative programming languages, the applications of high-level automatic parallelisation in such languages remains largely unexplored.

Literature on high-level automatic parallelisation dates back to 1970s [2] — describing how to perform parallelisation in the presence of effectful code [3] and aliasing [4]. However, despite extensive research, the lack of practical implementations (which integrate into existing workflows) has led to slow adoption rates of this technology. This project aims to provide a practical implementation of high-level automatic parallelisation, while exploring its potential in imperative programming languages.

1.1 Project Summary

In this project, I designed a *mid-featured, sequential, imperative programming language* with an *accompanying high-level automatic parallelisation compiler*. My language (hereby dubbed *Kautuka*, after the Indian ritual *thread*) implements a sensible subset of Go with minor syntactic changes. The compiler automatically compiles (sequential) Kautuka to parallel Go code, which can be executed using Go's compiler.

I theorised that high-level automatic parallelisation could improve the performance of I/O-bound programs. This drove my decision to use Go as the compilation target, as it provides robust support for I/O operations and efficient parallelisation primitives. Since Kautuka compiles straight to Go, we can integrate Kautuka into existing Go codebases without disrupting existing workflows. However, much of this work is language agnostic, and generalises to other imperative programming languages beyond Go.

High-level parallelisation in imperative languages introduces complexities not commonly found in toy functional languages (such as variable mutation, for-loops, and imperative-style functions). This project addresses both the theoretical and practical challenges presented by these issues.

The focus of this dissertation is on designing, implementing, and evaluating a software engineering project. However, I also decided to explore three research questions to help inform my requirements analysis (section 2.7) and form the basis of my evaluation (section 4):

- *Does high-level automatic-parallelisation theory, previously developed for toy functional languages, transfer across to a full-featured imperative language?* In sections 2.4 to 2.6, I explain the theoretical extensions needed to make this transition, and in section 3.5 I describe **a novel approach to cost analysis (execution time estimation) to overcome constraints found in imperative languages.**

- *Can I/O-bound programs benefit from high-level automatic parallelisation?* In section 2.4, I design the theory required for I/O parallelisation. Section 4.3 demonstrates that **Kautuka's performance exceeds that of equivalent sequential Go code for I/O-bound programs**.

- *Are there real-world applications of this compiler?* In section 3.5, I implement a powerful cost-inference algorithm to minimise user annotations. This improves the accessibility of this language to the average programmer. In section 4.4 I demonstrate how **Kautuka can be integrated into existing Go codebases with minimal effort.**

Chapter 2

Preparation

High-level automatic parallelisation utilises *effect systems* to produce *safe* and *efficient* parallel code. Parallel code is *safe* if it avoids race conditions (such as the concurrent execution of `x = 1` and `x = 2`), and *efficient* if it outperforms the original sequential code. Effect systems augment classical type systems with *effects*, to reason about program execution behaviours such as side effects and execution time. Similar to type systems, effect systems consist of *typing judgements* which infer expression properties — the execution time of `e1;e2` can be inferred to be the sum of `e1` and `e2`'s execution times.

In sections 2.4 to 2.6, I outline three effect systems required for my implementation of automatic parallelisation. *Side-effect systems* (section 2.4) prove that parallelisation is safe while *type-cost* (section 2.5) and *runtime-cost systems* (section 2.6) ensure that produced parallel code is efficient. These systems form the basis of my compiler implementation (sections 3.4 and 3.5). Section 2.7 details the project requirements and the software development practices followed throughout the project.

2.1 Project Overview

The project consists of two components: the Kautuka language and the high-level automatic parallelisation compiler (hereby referred to as just *the compiler*). This section describes the compilation process of Kautuka, and how Kautuaka can be integrated into existing Go codebases (section 2.1.1). Section 2.1.2 outlines Kautuka's key language features, and section 2.1.3 describes each step of the compilation process (and the purpose of each effect system).

2.1.1 Compilation of Kautuka

Kautuka's compiler is written in OCaml (ML); compiling Kautuka (Kau) into parallelised Go code. The produced Go code is then compiled into executable machine code using Go's existing compiler. The components implemented in this project are highlighted in grey on the tombstone diagram (fig. 2.1).

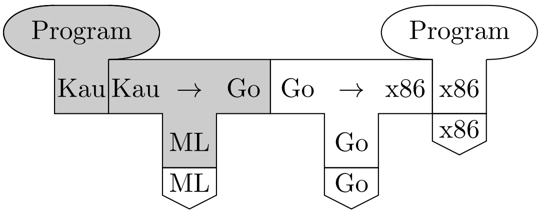

Figure 2.1: Kautuka Compilation

To integrate Kautuka into existing Go codebases, we expect the programmer to identify (sequential) Go files containing large, independent computations. These files can be translated into Kautuka by hand, as the languages share similar syntax. The programmer marks groups of code representing a single *task* with curly braces (which we refer to as *code blocks*). Our compiler then compiles Kautuka programs back to parallelised Go code, placing them alongside the Kautuka files. The Go codebase now contains parallelised Go files, and can be executed as normal using the standard Go compiler.

2.1.2 Kautuka Language

Kautuka implements a sensible subset of Go (with minor syntactic changes), and closely resembles other C-like languages. This section provides a brief overview of all key language features.

Kautuka has the following types:

$$\tau ::= \mathit{int},\ \mathit{bool},\ \mathit{string},\ \mathit{unit},\ \mathit{file\_ref},\ \tau_1 * \cdots * \tau_n \to \tau$$

where $\tau_1 * \cdots * \tau_n \to \tau$ is the imperative-style function signature. Kautuka does not support currying or higher-order functions. Although I planned to implement recursion as a stretch goal, I was unable to complete this extension due to time constraints.

Kautuka does not support exponentiation, division, or modulus operations. Extending the language and compiler with division and modulus is relatively straightforward, however the exponentiation operator would be limited to only integer exponents[1]. Kautuka includes a subtraction operator, but does not support negative numbers — the responsibility is on the programmer to ensure that the result of subtraction is non-negative.

The language contains inbuilt console I/O functions: *input* and *print*, and file-I/O functions: *open*, *read*, *write*, and *append*. File references (of type *file_ref*) are obtained by calling the *open* function with a filename string, and can be used to read and write to files. Disjoint file references are tracked using basic aliasing analysis (section 3.4), this project restricts aliasing analysis to just file references.

Kautuka also contains variables, control flow, functions, and code blocks (also referred to as just *blocks*). Variables are declared with `x := e` and are assigned to with `x = e`. Variables declared in a local scope are bound to that scope. In Kautuka, variable types are inferred and so do not need to be annotated. *For-loops* always take the form `for x := e1; x < e2; x++ { e3 }`, so that the number of loop iterations can be consistently estimated during cost analysis (section 3.5). Kautuka also supports for-each statements `for x := range str { e }`, iterating over every character of the string `str`. Functions are defined with `func f(param_1 t_1, ..., param_n t_n) t { e }`, where `t` is the function's return type and `e` contains at least one `return` statement (if the return type is *non-unit*). Kautuka contains expressions (e.g., `1 + 2`), commands (e.g., `x := 1`), and structures (e.g., `if true { 1 } else { 2 }`). However, we treat commands and structures as unit-type *expressions* to simplify our typing rules.

2.1.3 Automatic Parallelisation Pipeline

This section outlines the contributions of each effect system to the compiler pipeline. Detailed descriptions of each effect system are provided later in this chapter.

Side-effect tracking *statically* analyses code blocks, to determine which side effects they may potentially produce. Side effects are *non-interfering* if they can be performed in parallel without causing a race condition. If two blocks contain non-interfering side effects, then their sequential execution behaviour is equivalent to all possible parallel execution behaviours — proving that the parallelisation is safe. Note that side effects produced by local variables bound to a scope are ignored outside that scope.

[1] A decision explained in appendix C.

```
// No side effects
func f(a int) int {
  return a + 1
}

// Side effects = { Write : Console } ← (x, y local)
func g(x int, y int) int {

  // Side effects = { Read : Var(x), Write : Console } ← (z local)
  {
    z := f(x)
    print(z)
  }

  // Side effects = { Read : Var(x), Write : Var(y) }
  {
    y = x + 10
  }

  // These side effects are non-interfering
  // So it is safe to parallelise these blocks
}
```

Cost analysis determines whether parallelisation produces more *efficient* code. This is achieved by estimating the runtimes of both the *parallel* and *sequential* forms of code. The first step is to estimate the *size* of data types using type-cost analysis. The size of an integer is its numerical value, which can be estimated by tracking its upper and lower bounds. However, finding the output sizes of functions is challenging as they dependent on the function's input sizes, which vary by call site. To avoid tracing function calls throughout the program (which blows up inference time exponentially), we instead generate mappings (also known as *function summaries*) from input sizes to output sizes for each function. At each function call, we look up this mapping and pass in the input sizes to obtain the output size. Since this example is simple, we provide exact values for data-type sizes, as opposed to upper and lower bound estimates:

```
// f_size(a) = a + 1 (maps input size to output size)
func f(a int) int {
  return a + 1
}

func g(x int, y int) int {

  {
    z := f(x)   // Size of z = f_size(x) = x + 1
    print(z)
  }

  {
    y = x + 10  // Size of y = x + 10
  }

}
```

Using this information, we can estimate the *runtime* of each block using a runtime-cost system. In this dissertation we use *runtime* to refer to the time taken to execute an expression, and *dynamic* to refer to analysis performed during program execution. Similar to before, we generate mappings from input sizes to function runtimes; a function's runtime depends on the size of its inputs. We generate *runtime-cost expressions* by summing together runtimes in a code block. These expressions are *dynamically evaluated* once input sizes are known, to produce accurate runtime estimates. If we assume that `print(x)` takes $(100a)\mu s$ where a the size of x, and $x + y$ takes $(2a + 2b)\mu s$ where a and b are the sizes of x and y, and all remaining instructions take no time, then we produce the following runtime-cost expressions (in microseconds):

```
// f_runtime(a) = 2a + 2 (maps input size to runtime)
func f(a int) int {
  return a + 1
}

func g(x int, y int) int {

  // Runtime expression = 102x + 102
  {
    z := f(x)    // Runtime = f_runtime(x) = 2x + 2
    print(z)     // Runtime = 100 * f_size(z) = 100(x + 1) = 100x + 100
  }

  // Runtime expression = 2x + 20
  {
    y = x + 10  // Runtime = 2x + 2(10) = 2x + 20
  }

}
```

Functions in Go can be marked as *goroutines*[2] using the `go` keyword. When executed, a goroutine runs in the background, allowing execution of the main program to continue. Goroutines are lightweight green threads managed by the Go runtime, which are cheap to spawn and run as they do not interact with the underlying OS. Goroutines are a form of multicore parallelism: goroutine threads automatically execute across multiple cores if available. To run *code blocks* in parallel, we wrap them in anonymous functions, mark them as a goroutines, and call the functions. For example `{ print(x) }` would become `go func() { print(x) }()`.

Code blocks are parallelised if it is *safe* and produces more *efficient* code. Both side-effect tracking and the generation of *runtime-cost expressions* can be performed statically. However, runtime-cost expressions can only be *evaluated* dynamically. Hence, *safety* can be proved at compile time, but *efficiency* cannot. The solution is to parallelise code into both its sequential and parallel forms if it is *safe*. During the program's execution, we can evaluate the runtime-cost expressions and pick which form to execute based on the result. If $\mathit{sequential\_runtime} < \mathit{parallel\_runtime} + \mathit{parallelisation\_costs}$ then we execute the sequential form, else we choose the parallel form. $\mathit{parallel\_runtime}$ is the greatest individual runtime of all parallelised blocks, and $\mathit{parallelisation\_costs}$ account for the extra cost of spawning threads and communicating across processors. Here we assume that parallelisation costs sum to $50\mu s$:

[2] https://go.dev/tour/concurrency/1

```
func f(a int) int {
  return a + 1
}

func g(x int, y int) int {

  // if sequential_runtime < parallel_runtime + parallelisation_costs
  if (102x + 102) + (2x + 20) < max(102x + 102, 2x + 20) + 50 {

    // sequential execution
    {
      z := f(x)
      print(z)
    }

    {
      y = x + 10
    }

  } else {

    // parallel execution
    go func() {
      z := f(x)
      print(z)
    }()

    go func() {    // spawn thread
      y = x + 10
    }()            // invoke thread

  }
}
```

Key Takeaway. *Our program analysis consists of two parts: side-effect tracking to determine if parallelisation is* ***safe****, and cost analysis to determine if produced parallel code is more* ***efficient*** *than the original sequential code. If the first criteria is met (statically), then we compile the code into its sequential and parallel form. If the second criteria is met (dynamically), we execute the parallelised form of code.*

2.2 Background Work

High-level automatic parallelisation literature tends to focus on either *side-effect tracking* or *cost analysis*. This section provides an overview of previous work, and describes how my project combines and builds upon these ideas.

Side-Effect Tracking

Lucassen and Gifford [3] first proposed a side-effect system, which conservatively infers side effects produced by each expression in a statically-scoped Lisp dialect. Subsequent papers build upon this work by adding more complex features to the language, such as first class functions [5] and aliased reference values [6]. This project extends side-effect systems in a different direction to most papers: by implementing this theory into an imperative language. Section 3.4 describes how we tackle the challenges posed by this new paradigm, such as imperative-style scoping rules.

Cost Analysis

Reistad and Gifford [7] describe how to estimate the runtime of expressions statically, through algebraic cost reconstruction. Algebraic cost reconstruction consists of: reconstruction, unification, and constraint solving in order to produce runtime bound estimates. This technique can be applied purely statically, however the disadvantages are that this technique often forms inaccurate bounds or non-convergent constraints.

I instead opted for the approach highlighted by Huelsbergen et al. [8], using hybrid analysis (a combination of static and dynamic analysis) to guide parallelisation. *Runtime-cost expressions* are generated statically, and are evaluated dynamically to produce runtime estimates. To the best of my knowledge, this has never before been implemented in an imperative language compiler. Similar solutions use dynamic profiling: analysing instruction execution time to inform parallelisation whilst the program is running. However, this is outside the scope of this project.

The primary challenge of this project will be designing my own theory to infer both data-type sizes and runtime estimates in an imperative language. Since this language is mid-featured, I develop algorithms to deal with mutable variables, for-loops, and I/O operations, which are not commonly found in literature (sections 3.5.1 and 3.5.2)

Key Takeaway. *Prior work has addressed the implementation of side-effect tracking and cost analysis in toy ML/Lisp dialects. This project extends these ideas by incorporating this analysis into an imperative programming language compiler. This requires the development of novel theory for cost analysis, to overcome issues not commonly found in functional language implementations.*

2.3 Type Systems

Type systems consist of rules (*typing judgements*), describing how *types* are assigned to syntactic expression constructions. Standard type systems constrain the form of expressions based on their type — guaranteeing the absence of runtime type errors. Types, and type systems, can be enriched with *effects* to describe an expression's execution behaviour, such as its runtime and side effects.

2.3.1 Typing Rules

Standard type systems contain typing judgements of the form $\Gamma \vdash e : \tau$. This is read as: *expression e has the type τ in the typing context* Γ. The typing context Γ is a partial function, mapping variables to their associated types in the current environment ($\Gamma : \mathit{Var} \to \tau$). The notation $\Gamma[x : \tau]$ asserts that the typing context Γ must contain the mapping $x \mapsto \tau$.

Typing relationships are defined inductively with *typing rules*:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma_k \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}\ (\textit{rule-name})$$

This can be read as: *if the premises above the bar* $(\Gamma_i \vdash e_i : \tau_i)$ *hold for all* $0 \leq i \leq k$*, then the conclusion below the bar* $(\Gamma \vdash e : \tau)$ *must also hold.*

2.3.2 Effect Systems

Computational effects (or just *effects*) describe the runtime behaviour of expressions, beyond standard typing guarantees. Expression effects are either *immediate* or *latent*. Consider the following code example:

```
// e1
print("hello world")

// e2
func g() {
  input()
}
```

Immediate effects are produced when an expression is evaluated, for example `e1` produces the immediate side effect $\{\textsc{write} : \text{console}\}$. Latent effects are effects encapsulated by functions, and are only produced when the function is called. Expression `e2` defines a function `g` and produces no immediate side effects (written as $\{\}$). However, function `g` contains effectful code which produce the side effect $\{\textsc{read} : \text{console}\}$[3] when called. This side effect is the *latent effect* of function `g`.

Base types τ refer to a program's standard types: *int, string, bool,* etc. Enriching base types with an arbitrary effect eff produces *effectful types*, written as $\tau^{\text{eff}}, \text{eff}$. The eff component refers to the immediate effects produced by an expression. The τ^{eff} component represents base types τ enriched with latent effects: all functions in τ are annotated with latent effects. A function's latent effect is written above the arrow in the function's signature.

For example:

$$\text{Base type } (\tau) \text{ of e1: } ()$$
$$\text{Effectful type } (\tau^{\text{eff}}, \text{eff}) \text{ of e1: } (), \{\textsc{write} : \text{console}\}$$

$$\text{Base type } (\tau) \text{ of e2: } g : () \to ()$$
$$\text{Effectful type } (\tau^{\text{eff}}, \text{eff}) \text{ of e2: } g : () \xrightarrow{\{\textsc{read} : \text{console}\}} (), \{\}$$

Effect-system typing judgements take the form $\Gamma^{\text{eff}} \vdash e : \tau^{\text{eff}}, \text{eff}$. The typing context Γ^{eff} maps variables to their base types now enriched with latent effects $(\Gamma : \textit{Var} \to \tau^{\text{eff}})$.

Key Takeaway. *Type systems can be extended with effects to produce* ***effect systems****, providing richer analysis of program execution behaviour. Effect system typing judgements take the form* $\Gamma^{\text{eff}} \vdash e : \tau^{\text{eff}}, \text{eff}$*. The* τ^{eff} *component represents an expression's* ***immediate*** *effect and* τ^{eff} *represents an expression's base type enriched with* ***latent effects****.*

[3] In reality `input()` also writes to the console, but for the sake of this example we assume that it does not.

2.4 Side-Effect Analysis

Certain operations, such as I/O, do not fit into the typical lambda-calculus style evaluation. These operators are referred to as *side-effecting operations*, producing *side effects* when evaluated. This section explores how we define side effects in Kautuka (section 2.4.1) and how we use them to build a side-effect system (section 2.4.2).

2.4.1 Side Effects

Some considerations must be taken when defining side effects, as they are not well-defined in literature — most notably the treatment of variable read and writes. In the following example:

```
{
  x := 0
  x = 1
}
```

The code block defines (`:=`) a variable `x` and mutates it (`=`). Mutating a variable produces a WRITE side effect. However, since `x` is local to the block, this effect is encapsulated: all changes to the local variable `x` cannot be observed outside the block. In this project, we choose to ignore non-observable side effects by removing local-variable side effects at the end of every scope.

However, non-local variable side effects cannot be ignored as they may produce *data races*. For example, if one block writes to a variable and a second block reads or writes to that same variable in parallel, this causes a *race*. Hence, we are required to track non-local variable *read* and *write* side effects.

Console and file I/O are also tracked in this project. However, file I/O requires aliasing analysis in order to identify disjoint file references (section 3.4.1).

Side effects comprise two components: a *channel* and an *operation*. A *channel* describes what a side effect interacts with, in our case: non-local variables, console I/O and file I/O. It is atypical to consider non-local variables as a channel. However, since we treat non-local variables and channels in the same way, describing non-local variables as a channel simplifies our definitions. *Operations* describe the interaction with the channel, this project only considers *read* and *write* operations[4], written as R and W respectively.

$$\begin{aligned} \textit{Operation} &= \{\ \textsc{r}, \textsc{w}\ \} \\ \textit{Channel} &= \{\ \text{console}, \text{var}(x), \text{file}(\textit{id}, \textit{ref})^{5}\ \} \\ \textit{Side-Effect} &= \textit{Channel} \times \textit{Operation} \end{aligned}$$

The expression `print("hello")` produces the side effect $(\textsc{w}, \text{console})$, hereby written with the notation $\textsc{w} : \text{console}$. Expressions can produce multiple side effects, which we represent with a *side-effect set* f, where $f \in \mathscr{P}(\textit{Side-Effect})$. For example, the expression `print(x)` produces the side-effect set $\{\textsc{r} : \text{var}(x), \textsc{w} : \text{console}\}$. A set is sufficient for our analysis, as we are not concerned with the ordering of side effects.

[4] Initialisation operations are also common in literature, however these are not necessary for our analysis as local-variable side effects are ignored (variable initialisations are dropped once we leave a scope).

[5] The values *id* and *ref* are required for aliasing analysis (section 3.4.1).

2.4.2 Side-Effect System

A *side-effect system* extends the standard type system with side effects (se), to track potential side effects produced by each expression. The *immediate effect* is the set of side effects $f \in \mathscr{P}(\textit{Side-Effect})$ produced by an expression's evaluation. Functions are labelled with their *latent side effects* by enriching base types τ to produce effectful types τ^{se}. From this, we derive the side-effect typing judgement $\Gamma^{\text{se}} \vdash e : \tau^{\text{se}}, f$. A subset of Kautuka's side-effect system typing rules are listed below, with explanations provided in section 3.4.2. τ^{se} and Γ^{se} are written as τ and Γ for brevity:

$$\frac{}{x : \tau,\ \Gamma \vdash x : \tau, \{\textsc{r}, \text{var}(x)\}}\ (\textit{var-read})$$

$$\frac{\Gamma \vdash e_1 : \tau_1, f_1 \qquad (x : \tau_1), \Gamma \vdash e_2 : \tau_2, f_2}{\Gamma[x : \tau_1] \vdash x = e_1;\ e_2 : \tau_2, (f_1 \cup f_2 \cup \{(\textsc{w}, \text{var}(x))\})}\ (\textit{var-assign})$$

$$\frac{\Gamma \vdash e : \textit{string}, f}{\Gamma \vdash \text{print}(e) : \textit{unit}, f \cup \{(\textsc{w}, \text{console})\}}\ (\textit{print}) \qquad \frac{\Gamma \vdash n : \textit{int}, \{\}}{\Gamma \vdash \text{input}(n) : \textit{string}, \{(\textsc{w}, \text{console})\}}\ (\textit{input-1}[6])$$

$$\frac{\Gamma \vdash e : \textit{file\_ref}(i, g), f \qquad \Gamma \vdash n : \textit{int}, \{\}}{\Gamma \vdash \text{read}(e, n) : \textit{string}, f \cup \{(\textsc{r}, \text{file}(i, g))\}}\ (\textit{file-read-1}[6])$$

$$\frac{\Gamma \vdash e_1 : \textit{file\_ref}(i, g), f_1 \qquad \Gamma \vdash e_2 : \textit{string}, f_2}{\Gamma \vdash \text{write}(e_1, e_2) : \textit{unit}, (f_1 \cup f_2 \cup \{(\textsc{w}, \text{file}(i, g))\})}\ (\textit{file-write})$$

Key Takeaway. *Side-effect systems are used to track all potential side effects produced by an expression. The side-effect typing judgement takes the form $\Gamma^{se} \vdash e : \tau^{se}, f$, to describe both immediate and latent side effects. If the side effects of two code blocks are* ***non-interfering****, then it is* ***safe*** *to run them in parallel.*

2.5 Type-Cost Analysis

Program analysis techniques often calculate numerical properties associated with an expression or type, known as a *costs.* This section lays the groundwork for constructing effect systems which estimate both *type costs* (data-type size estimates) and *runtime costs* (runtime estimates). *Cost bounds* (upper and lower bounds on a cost) provide sufficiently accurate estimates for costs. A cost bound indicates a cost's order of magnitude — knowing whether a block's runtime is in the order of $1\mu s, 1ms$ or $1s$ is sufficient to tell us if the block should be parallelised in the majority of cases.

As mentioned in section 2.1.3, we want to avoid tracing function calls throughout the program when analysing function costs. Since a function's cost is dependent on its input sizes, mappings can be generated from input sizes to function costs for each function in the program. At every function call, we look up this mapping and pass in the input sizes to obtain its cost.

2.5.1 Type-Cost Bounds

Cost bounds consist of a lower bound l and an upper bound u, written as $\langle l, u\rangle$. Let us consider the following example:

[6] The extra argument n is required for type-cost analysis (section 3.5.1).

```
if cond {
  x = 3
} else {
  x = 5
}
```

The size of `x` at the end of the *if-statement* is represented with the bound $\langle 3, 5 \rangle$. We remember that the size of an integer is its numerical value. In the case of integers, *type costs* refer to both the type *and* size of an expression; the type cost of x would be written as $int\langle 3, 5 \rangle$. This means that, no matter which branch is executed, `x` will always be integer bounded by $\langle 3, 5 \rangle$. If we subsequently apply `x = x + 1`, then `x`'s new type cost will be $int\langle 3 + 1, 5 + 1 \rangle = int\langle 4, 6 \rangle$. These inference rules are applied statically to generate type costs for all expressions. Section 3.4.2 describes how we deal with complications that arise in the presence of control flow.

If a program were to be absent of functions, then all cost bounds would be *concrete* (bounds would only contain integer values). However, the introduction of functions requires us to generate mappings from input costs to function costs. In this case, *input costs* refer to the size of the function's inputs, and *function costs* refers to the size of the function's output. A function call's type cost can be derived from its output size and the return (base) type of the function's signature. Considering the following example:

```
func g(y int) int {
  z := y + 1
  return z
}
```

The type cost of `z` is $int\langle y_l + 1, y_u + 1 \rangle$, where y_u and y_l refer to the upper and lower bounds of y. Note that y here represents the size of the function's input, which is only known at execution time. Since `z` is returned by the function, the function's return size is $\langle y_l + 1, y_u + 1 \rangle$, which can be described with the mapping $g_{size}(y) = \langle y_l + 1, y_u + 1 \rangle$ (mapping input sizes to output sizes).

If we were to call this function with `g(5)`, where 5 has the size $\langle 5, 5 \rangle$, we would look up the mapping and substitute in the input size. This produces the output size $g_{size}(\langle 5, 5 \rangle) = \langle 6, 6 \rangle$, hence the function call has type cost $int\langle 6, 6 \rangle$. If type costs are treated as effects, then the mapping from input to output sizes for functions can be thought of as the function's latent effect — this idea is explored further in the next section.

In the case above, we describe the type cost $int\langle y_l + 1, y_u + 1 \rangle$ as being dependent on the unknown variable y. To represent this mathematically, we introduce a *dependent cost calculus*, detailed in appendix C. However to summarise: costs are represented as a polynomial, whose variables represent unknown input sizes.

2.5.2 Type Costs

Data-type *sizes* are metrics used to estimate the runtime of operators, in-built functions and control flow. If the runtime of string concatenation depends on the length of its input strings, then string length is a useful size metric for strings. In Kautuka, the size of integers are their *numerical values*, and the size of strings are their *lengths*. Sized types τ_{sized} are types where size is well-defined, in our case *ints* and *strings*. Whereas *unsized types* τ_{unsized} do not have an associated size: *unit*, *bool* and *file_ref*. Functions are sized if and only if their return type is sized. Types which may either be sized or unsized are written as τ.

A *quantified type* is a combination of a *sized type* and its *associated size*. *Type costs* collectively describe both quantified types and unsized types. Note that in the previous section, the definitions of quantified types and type costs were equivalent as we were only considering sized types. Quantified types are written as $\tau_{\text{sized}}(c)$ or $\tau_{\text{sized}}\langle l, u\rangle$, where c and $\langle l, u\rangle$ are the type's size (in cost bound form).

$$\begin{aligned}
\textit{Sized-Type} &= \{\textit{int, string, } \tau_1 * \cdots \tau_n \to \tau_{\text{sized}}\} \\
\textit{Unsized-Type} &= \{\textit{unit, bool, file\_ref, } \tau_1 * \cdots \tau_n \to \tau_{\text{unsized}}\} \\
\textit{Quantified-Type} &= \textit{Sized-Type} \times \textit{Cost-Bound} \\
\textit{Type-Cost} &= \textit{Quantified-Type} \cup \textit{Unsized-Type}
\end{aligned}$$

A type-cost system is an effect system used to infer the type-cost effects (cost) of all program expressions. An expression's type cost is its *immediate effect*, and the mapping from function input to output sizes is its *latent effect*. For example, the latent effect of function g from the previous section, $g_{size}(y) = y + 1$, is written as:

$$g : int \xrightarrow{g_{size}(y)=y+1} int$$

When calling a function, we only look up this mapping if the function's return type is *sized*. To simplify our typing rules, mappings contain all function input variables (including unsized inputs), however the output can only depend on sized inputs.

Effect system judgements generally take the form $\Gamma^{\text{eff}} \vdash e : \tau^{\text{eff}}, \text{eff}$. However, the type-cost system is an exceptional case. Typically, a typing context Γ^{eff} maps variables to their base types, and functions to their function signatures annotated with latent effects. However, Γ^{cost} instead maps variables to their *type costs* (base types annotated with immediate effects). This occurs because variable types have a size, and so variables also store effects (which is uncommon in effect systems). Since τ^{cost} is enriched with both immediate and latent effects, the immediate effect component (cost) is no longer required on the right side of the judgement ($\tau^{\text{cost}}, \text{cost}$). So typing judgements for type-cost systems are written in the form $\Gamma^{\text{cost}} \vdash e : \tau^{\text{cost}}$.

In a type-cost system, types should be marked as τ^{cost} (a sized or unsized type), $\tau^{\text{cost}}_{\text{sized}}$ (a sized type) or $\tau^{\text{cost}}_{\text{unsized}}$ (an unsized type) — where sized types can have an associated size. However, this notation is very verbose for typing rules, so we instead write these as $\tau, \tau_{\text{sized}}$, and τ_{unsized}. Similarly, typing contexts Γ^{cost} are written as Γ. If we want to specify that a type is a base type (what would originally have been written as τ), we mark this as τ^{base}. Note that τ_{sized} and τ^{base} refer to the *same* base type τ, with the former enriched with cost effects.

A subset of type-cost system typing rules are provided below, with full derivations described in section 3.5.1:

$$\frac{}{\Gamma \vdash n : int\langle n, n\rangle}\,(int) \qquad \frac{}{\Gamma \vdash b : bool}\,(bool) \qquad \frac{\text{len}(s) = n}{\Gamma \vdash s : string\langle n, n\rangle}\,(string)$$

$$\frac{\Gamma \vdash e_1 : int(c_1) \quad \Gamma \vdash e_2 : int(c_2)}{\Gamma \vdash e_1 + e_2 : int(c_1 + c_2)}\,(add) \qquad \frac{\Gamma \vdash e_1 : int(c_1) \quad \Gamma \vdash e_2 : int(c_2)}{\Gamma \vdash e_1 * e_2 : int(c_1 \cdot c_2)}\,(mult)$$

(*def-func-2*):

$$\frac{x_1 : \tau_1(c_1), \ldots, x_n : \tau_n(c_n), \Gamma \vdash e : \tau_{\text{sized}}(c) \qquad (g : \tau_1^{\text{base}} * \cdots * \tau_n^{\text{base}} \xrightarrow{g_{size}(c_1,\ldots,c_n)=c} \tau^{\text{base}}), \Gamma \vdash e' : \tau'}{\Gamma \vdash (\text{def } g(x_1 : \tau_1^{\text{base}}, \ldots, x_n : \tau_n^{\text{base}})\ \tau^{\text{base}}\ \{\, e\}; e') : \tau'}$$

(*apply-func-2*):

$$\frac{\Gamma \vdash g : (\tau_1^{\text{base}} * \cdots * \tau_n^{\text{base}}) \xrightarrow{g_{size}(c_1,\ldots,c_n)=c} \tau^{\text{base}} \qquad \Gamma \vdash e_1 : \tau_1(c_1) \quad \cdots \quad \Gamma \vdash e_n : \tau_n(c_n)}{\Gamma \vdash g(e_1,\ldots,e_n) : \tau_{\text{sized}}(g_{size}(c_1,\ldots,c_n))}$$

Key Takeaway. *Cost bounds estimate numerical properties of either types or expressions. An example is* ***type costs****, which estimate data-type sizes. Function type costs are represented by producing a mapping from the function's input sizes to its output size. At each function call, we substitute in the input sizes to obtain the function's output size (if the return type is sized).*

2.6 Runtime-Cost Analysis

Runtime-cost analysis builds upon type-cost analysis to estimate expression runtimes. This process consists of two stages: we first estimate the runtime of instructions (as a function of their input sizes) using static profiling (section 2.6.1). These estimates are then used to infer the runtime costs of each expression, in a similar fashion to type-cost analysis (section 2.6.2).

2.6.1 Instruction Runtime Estimation

We collectively refer to operations and initialisations of control flow structures as *instructions*. The execution time of operations may depend on their input sizes. For example, multiplication of two integers x and y may take $(a+b)\mu s$ where a and b are the sizes of x and y — evaluating `3 * 5` would take $15\mu s$. By executing instructions on various input sizes and measuring their runtimes, we derive the relationship between *input size* and *runtime* for each operation — in process called *static profiling*. Initialisation times for control flow structures are modelled as *constant*, whose values are also obtained through static profiling. We capitalise instruction names when describing the mapping from input sizes to runtimes (control flow structure initialisations take no inputs). Using the above example, the multiplication operator would have the mapping $\text{MULT}(x,y) = x + y$. If calling a function incurs a constant cost of $50\mu s$, this is written as $\text{FUNC\_CALL}() = 50$.

Similar to type-costs, we represent runtime-cost estimates using *cost bounds*. If we were to execute `x * y`, where the sizes of `x` and `y` are $\langle 1,2\rangle$ and $\langle 5,6\rangle$ respectively, then we would obtain the runtime cost $\text{MULT}(\langle 1,2\rangle,\langle 5,6\rangle) = \langle 1,2\rangle + \langle 5,6\rangle = \langle 6,8\rangle\mu s$.

2.6.2 Code-Block Runtime Estimation

By reasoning about control flow and function calls, we can use runtime estimates for operators to produce *runtime-cost expressions*. For example, if an expression `e` takes t microseconds to execute, then `for i := 0; i < 10; i++ { e }` takes approximately $10t$ microseconds to execute. However, operation runtimes may depend on the loop's construction. If the runtime of `print(i)` depends on the size of i, then each iteration of `for i := 0; i < 1000; i += 100 { print(i) }` has a different runtime. We design algorithms to deal with such issues in sections 3.5.1 and 3.5.2.

For each function, we can derive a mapping from *input sizes* to *runtime cost*. For example, calling the following function:

```
func g(x int, y int) {
  return x * y
}
```

Takes $\text{MULT}(x, y) + \text{FUNC\_CALL}()$ microseconds, where $\text{FUNC\_CALL}()$ encapsulates extra costs incurred by invoking a function. If $\text{MULT}(x, y) = x + y$ and $\text{FUNC\_CALL}() = 10$, then we generate the following mapping: $g_{runtime}(x, y) = x + y + 10$.

Runtime-cost systems extend *type-cost systems* with *runtime effects*. The *immediate effects* are the expression's runtime, represented with the cost bound r. The *latent effects* are the function's mappings from input sizes to runtime costs. We enrich type-costs τ^{cost} with runtime effects run to produce τ^{run}, for example:

$$g : \mathit{int} * \mathit{int} \xrightarrow[]{\substack{g_{size}(x,y)=x\cdot y \\ g_{runtime}(x,y)=x+y+10}} \mathit{int}$$

Since the type τ^{run} is an extension of τ^{cost}, we still maintain the notion of sized and unsized types. From this, we derive the typing judgement $\Gamma^{\text{run}} \vdash e : \tau^{\text{run}}, r$. A subset of typing rules are shown below, with explanations provided in section 3.5.2.

Similar to before, we write the types τ^{run}, $\tau^{\text{run}}_{\text{sized}}$, $\tau^{\text{run}}_{\text{unsized}}$ as τ, τ_{sized}, τ_{unsized}. Γ^{run} is written as Γ and base types τ are specified with τ^{base}.

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1), r_1 \quad \Gamma \vdash e_2 : \mathit{int}(c_2), r_2}{\Gamma \vdash e_1 + e_2 : \mathit{int}(c_1 + c_2), r_1 + r_2 + \text{ADD}(c_1, c_2)} \ (\mathit{add})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1), r_1 \quad \Gamma \vdash e_2 : \mathit{int}(c_2), r_2}{\Gamma \vdash e_1 * e_2 : \mathit{int}(c_1 \cdot c_2), r_1 + r_2 + \text{MULT}(c_1, c_2)} \ (\mathit{mult})$$

$$\frac{\Gamma \vdash e_1 : \mathit{unit}, r_1 \qquad \Gamma \vdash e_2 : \tau, r_2}{\Gamma \vdash e_1; e_2 : \tau, r_1 + r_2} \ (\mathit{seq})$$

Note that in the following rules, g does not have a g_{size} mapping, as its return type is *unsized.*

(*def-func-1*):

$$\frac{x_1 : \tau_1(c_1), \ldots, x_n : \tau_n(c_n), \Gamma \vdash e : \tau_{\text{unsized}}, r \qquad (g : \tau_1 * \cdots * \tau_n \xrightarrow{g_{\text{runtime}}(c_1,\ldots,c_n)=r} \tau_{\text{unsized}}), \Gamma \vdash e' : \tau', r'}{\Gamma \vdash (\text{def } g(x_1 : \tau_1^{\text{base}}, \ldots, x_n : \tau_n^{\text{base}})\ \tau^{\text{base}}\ \{\, e\,\};\ e') : \tau', r'}$$

(*apply-func-1*):

$$\frac{\Gamma \vdash g : \tau_1 * \cdots * \tau_n \xrightarrow{g_{\text{runtime}}(c_1,\ldots,c_n)=r} \tau_{\text{unsized}} \qquad \Gamma \vdash e_1 : \tau_1(c_1), r_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n(c_n), r_n}{\Gamma \vdash g(e_1, \ldots, e_n) : \tau_{\text{unsized}}, (r_1 + \cdots + r_n + g_{\text{runtime}}(c_1, \ldots, c_n) + \text{FUNC\_CALL}())}$$

Key Takeaway. ***Static profiling*** *generates mappings from input size to runtime for each instruction. These mappings form the basis of our* ***runtime-cost system****, which statically generates* ***runtime-cost expressions*** *parameterised on the unknown size of function inputs.*

2.7 Requirements Analysis

Requirements for the core project are listed in the Success Criteria of the Project Proposal (Appendix F). These are summarised as: creating a high-level automatic parallelisation compiler for a mid-featured imperative programming language, implementing both effect tracking and cost analysis to guide parallelisation. Once these core criteria were met, I implemented stretch goals to explore my research questions — these informed my choice of extension features during requirements analysis.

1. *Transfer automatic-parallelisation theory from functional to imperative languages.*
    - Recursion: a feature commonly found in functional implementations
2. *Investigate the performance of high-level parallelisation on I/O-bound programs.*
    - Console I/O: Investigate performance on I/O operations
    - File I/O: Same as above, but with file-reference aliasing
3. *Investigate real-world applications of the compiler.*
    - Type-cost inference: minimise programmer overhead, for greater accessibility
    - Integration tools: integrate Kautuka into existing codebases (with minimal friction)

2.7.1 MoSCoW Analysis

To prioritise tasks, I performed MoSCoW[7] analysis for both language (table 2.1) and compiler (table 2.2) features. All core deliverables are *Must-Have* features, with extensions placed in the *Should-Have* and *Could-Have* categories. I prioritised extensions based on their relevance to my research questions, and their feasibility to complete within the project's timeframe.

MoSCoW Analysis of Kautuka Features:

MoSCoW Category	Kautuka Language Features
Must-have	Control flow (if-else, for-loops, functions), mathematical operations[8], boolean operations
Should-have	Console I/O (print, input), file I/O (open, read, write, append)
Could-have	Recursion, integration into Go
Won't-have	Structs, enums, modules

Table 2.1: MoSCoW analysis of Kautuka's language features

MoSCoW Analysis of Compiler Features:

MoSCoW Category	Automatic Parallelisation Compiler Features
Must-have	Non-local variable side-effect tracking, type-cost analysis (no inference), runtime-cost analysis, hybrid parallelisation
Should-have	I/O side-effect tracking, type-cost inference
Could-have	Recursive function type-cost inference
Won't-have	Algebraic cost reconstruction, dynamic profiling

Table 2.2: MoSCoW analysis of the compiler's features

[7] **M**ust have, **S**hould have, **C**ould have, **W**on't have.

[8] Excluding operations such as exponentiation and division.

2.7.2 Software Development Model

The implementation of this project was split into three chronological phases: implementing Kautuka's base compiler (excluding static analysis), implementing static analysis, and adding extensions. Implementing the base compiler for Kautuka naturally lends itself to spiral development: adding new syntax features and tests during each iteration. However, each static analysis stage depends heavily on the completion of the previous stage — making this phase more applicable to a waterfall approach. Despite its lack of flexibility, I found this approach to be very effective due to the project's fixed scope and deadline. For extensions, I returned to the spiral model: iteratively implementing extensions upon the existing compiler architecture.

2.7.3 Version Control and Tools Used

I used Git version control due to its powerful branching capabilities: allowing orthogonal tasks to be split across multiple branches. Once a task was complete, it could be merged back into `main`. This was useful during the spiral development phases of development where I could work on multiple features concurrently.

Instead of using existing benchmarking tools for *static analysis*, I decided to write my own custom scripts using bash. This enabled more precise control over which instruction runtimes I was measuring, and allowed me to analyse the effects of different compiler flags. I analysed the results, and hence derived mappings from input sizes to output runtimes, using Python's `sklearn` (model training) and `matplotlib` (data visualisation) libraries. I chose these libraries as they both act on `numpy` arrays, allowing me to train models and visualise the results without changing the data's representation.

2.8 Starting Point

I had no prior experience with OCaml, compilers or automatic parallelisation beyond the relevant Computer Science Tripos courses[9]. My previous experience with the Go programming language was limited to only basic syntax: mathematical operations, control flow and functions. Before starting, I conducted research with sample Go programs to investigate the feasibility of this project. Beyond that, I have no practical experience writing Go programs.

2.9 Summary

We introduce three effect systems to ensure that parallelisation produces *safe* and *efficient* parallel code, serving as the foundation to my compiler's implementation. We tackle challenges introduced by: functions, imperative-style scopes, and local variables, many of which only manifest in the context of imperative languages. My three research questions influenced requirements analysis, and I justify decisions made on the choice of tools and software development models used throughout the project.

[9] IA Foundations of Computer Science, IB Semantics, IB Compiler Construction, IB Concurrent and Distributed Systems, II Types, II Optimising Compilers.

Chapter 3

Implementation

The compiler implementation consists of **nine** pipeline stages. Section 3.1 provides an overview of the pipeline and my general approach to the compiler implementation. Subsequent sections (3.2 to 3.6) describe the implementation of each pipeline stage. The implementation builds upon theory established in the previous chapter, providing concrete typing rules for all effect systems. Section 3.4 extends the notion of side-effect tracking with *non-interference* operators and *aliasing analysis*. Sections 3.5 and 3.6 present a *powerful cost inference algorithm* to infer expression runtimes.

3.1 Implementation Overview

Kautuka's syntax and language design were formalised before development started on the project. The language specification can be found in appendix A. However, all key language considerations have already been outlined in the preparation chapter (section 2.1.2).

This section explores the compiler implementation: providing an illustration of the full compiler pipeline (section 3.1.1) and an overview of the code repository (section 3.1.2). Section 3.1.3 describes my general approach to implementing both the effect systems, and the remaining compiler stages.

3.1.1 Compiler Pipeline

The following diagram (fig. 3.1) summarises the compiler pipeline. We refer to each block in the flowchart as a pipeline *stage*, and each colour as a pipeline *phase*.

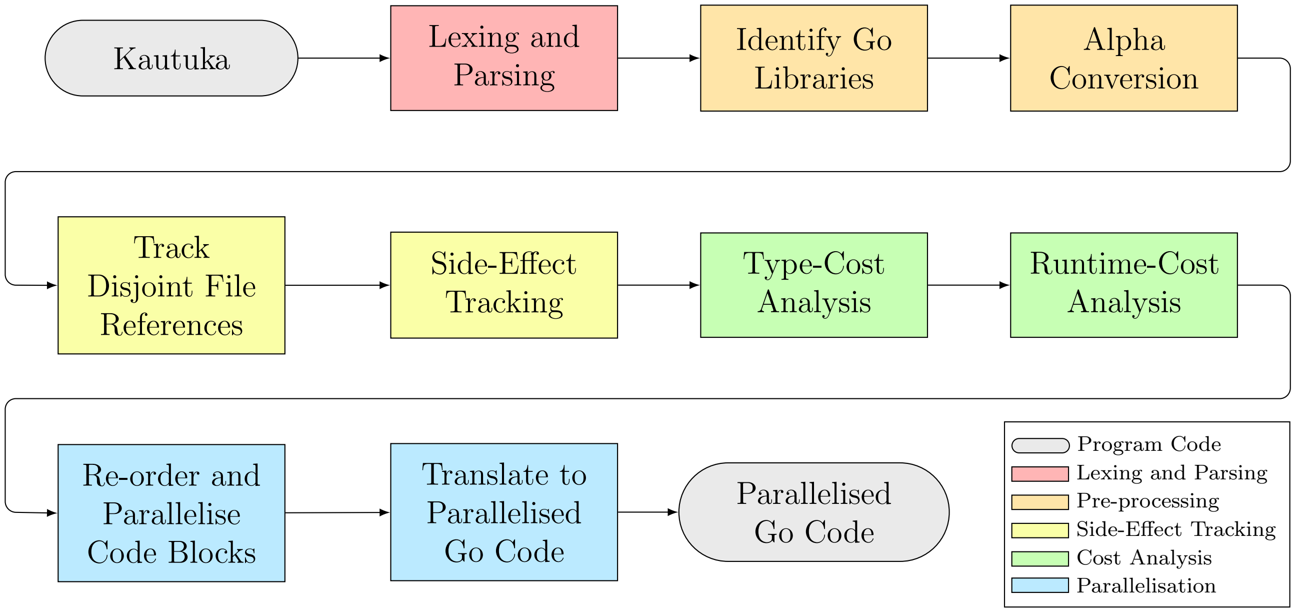

Figure 3.1: An overview of Kautuka's compiler pipeline

3.1.2 Repository Overview

The top-level directory contains the compiler entry point `src/bin/`, library files `src/lib/`, tests `tests/`, and benchmarks `benchmarks/`. The library directory includes folders `src/lib/util` (helper modules), and `src/lib/00_ast/` (AST type definitions and our *abstract analysis framework*, section 3.1.3). The remaining folders in this directory refer to each *phase* of the compiler pipeline. Static profiling and benchmarking are written in a combination of Python, Go and bash; all remaining code is written in OCaml. In total, I wrote **480 tests**: 432 unit tests and 48 end-to-end tests.

Folder	**Description**	**Lines**
`src/bin`	Compiler entry point, parses command line arguments	78
`src/lib/00_ast`	Contains AST type definitions and our abstract analysis framework	981
`src/lib/01_parsing`	Contains the lexer and parser	320
`src/lib/02_preprocessing`	Identifies libraries to import and performs alpha conversion	183
`src/lib/03_side_effect`	Tracks side effects for each code block	320
`src/lib/04_cost_analysis`	Estimates runtime of each code block	1120
`src/lib/05_parallelisation`	Converts AST into parallelised Go code	600
`src/lib/util`	Contains helper modules	496
`src/lib/static_profiling`	Estimates instruction runtime (as a function of input data-type sizes)	921
`tests`	Unit and end-to-end tests using `pytest`	1023
`benchmark`	Benchmarks for Kautuka performance	1154

Table 3.1: Repository overview. All code was written from scratch.

The OCaml compiler pipeline is built using the Dune build system [9]. Within each library, I group code into *modules*: a `.ml` file for each module body with a corresponding `.mli` interface file for type signatures. To maintain good coding practices, I followed OCaml's programming guidelines [10] and formatted my code with `ocamlformat` [11].

3.1.3 Implementation Approach

The project's implementation consists of three effect systems: a side-effect system, a type-cost system, and a runtime-cost system. Despite these systems having different effects and typing rules, they all share common generalisable features. All three effect systems implement a *post-order traversal* of an AST with an *accumulator environment*[1] (Γ^{eff}), a *monadic return type* (eff), and *typing rules.* The traversal maps an environment and AST expression

[1] In the case of type-cost analysis, the accumulator environment is not used.

to an effect-enriched environment (representing *latent effects*) and a monadic return type (representing *immediate effects*), $\mathit{Traversal}: \Gamma * \texttt{expr} \to \Gamma^{\text{eff}} * \text{eff}$.

In this project, traversing an AST with an accumulator and a monadic return type has uses beyond effect systems — the same pattern can be used to implement pipeline stages such as *alpha conversion*. Enabling the traversal to modify expressions ($\mathit{Traversal}: \Gamma * \texttt{expr} \to \Gamma^{\text{eff}} * \texttt{expr} * \text{eff}$) and generalising our definition of *effects* gives us an *abstract analysis framework*, which is utilised by almost[2] all stages of the pipeline.

Key Takeaway. *The compiler pipeline consists of phases, building upon the theory detailed in the Preparation chapter. Effect systems and other pipeline stages can be generalised into an* ***abstract analysis framework*** *by abstracting out common features.*

3.2 Lexing and Parsing

A Kautuka program is *tokenized* into a *token stream* using `ocamllex` [12]: a tool which generates lexers based on regular expression specifications. The token stream is *parsed* using `menhir` [13] to produce a program AST (in the form of OCaml algebraic data types). `menhir` compiles Kautuka's grammar specification (appendix A) into a parser. I opted to use parser generators, as opposed to handwriting my own parsers, as they pick up more errors, are easier to extend, and can handle complex LR(1) grammars.

Desugaring is also applied at this stage — the expression `x += e` is expanded to `x = x + e` and *if-elif-else* blocks are expanded to chained *if-else* blocks.

Key Takeaway. *The tools `ocamllex` and `menhir` produce lexers and parsers for our program based upon syntax specification. Lexers and parsers convert a program string into an AST, used in subsequent pipeline stages.*

3.3 Pre-processing

The *pre-processing phase* annotates and modifies the program AST to assist with future pipeline stages. We identify libraries which need to be imported in the produced Go code (section 3.3.1), and perform alpha conversion in preparation of side-effect tracking and cost analysis (section 3.3.2).

3.3.1 Identify Go Libraries

Unlike Go, Kautuka does not contain import statements or libraries, treating standard library functions as *in-built*. When compiling to parallelised Go, we traverse Kautuka using the *abstract analysis framework* to accumulate a list of required Go libraries. These import statements are inserted at the top of the produced Go code.

3.3.2 Alpha Conversion

One key distinction from functional languages is that imperative languages use *scoping blocks* as opposed to *let bindings*. Scoping blocks allow multiple local-variable declarations to be

[2] Excluding lexing, parsing, and translating the program into Go.

made inside expressions, whereas let bindings explicitly declare a single local variable at the expression's top level. We are required to identify local-variable declarations in scoping blocks for side-effect tracking (section 3.4.2). To do this, we traverse the expression with our abstract analysis framework and accumulate local variables. If a code-block expression $\{e\}$ contains local variables $x, y, \ldots, z$, we write this as $\{e\}_{\{x,y,\ldots,z\}}$.

The abstract framework can also be used to apply alpha conversion — a process of disambiguating logically distinct variables which share the same name. This is useful for both local-variable tracking and subsequent pipeline stages.

Key Takeaway. *Pre-processing utilises the **abstract analysis framework** to identify libraries, accumulate local variables defined in each block, and perform alpha conversion.*

3.4 Side-Effect Tracking

This section extends side-effect tracking with *aliasing analysis* (section 3.4.1) to track disjoint file references, providing a basis for file-I/O analysis. We use this analysis to produce *concrete typing rules* for Kautuka's side-effect system (section 3.4.2). Once side-effect sets have been inferred for each code block, a *non-interference operator* tells us if the side effects are non-interfering, and hence whether blocks can be *safely* parallelised.

3.4.1 File Tracking

Full reference analysis has been shown to be statically undecidable [14]. However, *aliasing analysis* provides a conservative approach to reference tracking, to determine whether references are *disjoint*. In our case, we use aliasing analysis guided by user annotations to track disjoint *file references*.

Two references are the *same* if initialised at the same location, and are *non-disjoint* if they point to the same physical file. The following example highlights the subtle differences between these definitions:

```
a := open("ref")
b := a
// a and b are the same and non-disjoint
// (the same reference stored in a is also assigned to b)

x := open("ref")
y := open("ref")
// x and y are different and non-disjoint
// (x and y's references were defined at different locations, however
//  they still point to the same file)
```

Existing literature uses *equality constraints* [15] or *union-find data structures* [16] to track direct and indirect references. However, this project does not require such complex algorithms as Kautuka only contains *direct* references. We instead mark reference with an *identifier* and *group*: the identifier tracks if references are the *same*, and groups use this information to track if references are *disjoint*.

A file reference is marked with a *fresh* (new) identifier on initialisation, which is passed along during variable assignment. A programmer often knows when they define disjoint references, however the program does not — hence we allow the user to optionally annotate disjoint references into the same *group* (represented with an *integer*). Unannotated references are assumed to be non-disjoint from all other references, and hence are placed into a *fresh* group. Note that this idea may be unintuitive, references known to point to *different* memory locations are placed into the *same* group. However, let us consider a system where references pointing to the *same* memory locations into the *same* group. Unannotated references would need to be placed into *every* group, as we cannot assume there are any memory locations it does not point to. This decision simplifies our typing rules, and reduces the burden on the programmer — successively defined file reference are more likely to be disjoint than the non-disjoint, reducing the number of groups required.

References in the same group are guaranteed to be disjoint, with the exception that multiple instances of the *same* reference can be in the same group; this is detected with the reference identifier. Hence, references are disjoint if they are in the *same group* and have *different identifiers.*

Reference tracking is implemented with the following typing rules. *file_ref*(i, g) is a file reference with identifier i and group g. We use i' to refer to a fresh identifier, and g' to refer to a fresh group.

$$\frac{\Gamma \vdash e : \mathit{string}}{\Gamma \vdash \mathrm{open}(e) : \mathit{file\_ref}(i', g')}\;(\mathit{open\text{-}1}) \qquad \frac{\Gamma \vdash e : \mathit{string} \qquad \Gamma \vdash n : \mathit{int}}{\Gamma \vdash \mathrm{open}(e, n) : \mathit{file\_ref}(i', n)}\;(\mathit{open\text{-}2})$$

$$\frac{\Gamma \vdash e_1 : \mathit{file\_ref}(i, g) \qquad (x : \mathit{file\_ref}(i, g)), \Gamma \vdash e_2 : \tau}{\Gamma \vdash x := e_1;\ e_2 : \tau}\;(\mathit{var\text{-}file\text{-}declare})$$

$$\frac{\Gamma \vdash e_1 : \mathit{file\_ref}(i_1, g_1) \qquad (x : \mathit{file\_ref}(i_1, g_1)), \Gamma \vdash e_2 : \tau}{\Gamma[x : \mathit{file\_ref}(i_2, g_2)] \vdash x = e_1;\ e_2 : \tau}\;(\mathit{var\text{-}file\text{-}assign})$$

(*var-file-decl*) and (*var-file-assign*) are captured by the traditional (*var-declare*) and (*var-assign*) typing rules if we consider file references with different identifiers and groups (e.g., *file_ref*(i_1, g_1) and *file_ref*(i_2, g_2)) to be of the same type.

3.4.2 Side-Effect Tracking

The following side-effect definitions were presented in the preparation chapter (section 2.4):

$$\begin{aligned} \mathit{Operation} &= \{\, \textsc{r}, \textsc{w} \,\} \\ \mathit{Channel} &= \{\, \mathrm{console}, \mathrm{var}(x), \mathrm{file}(i, g) \,\} \\ \mathit{Side\text{-}Effect} &= \mathit{Channel} \times \mathit{Operation} \end{aligned}$$

Two side effects are described as *non-interfering* if two expressions producing these effects have the same sequential and parallel execution behaviour. A *non-interference operator* # describes whether two *side effects* are non-interfering through *syntactic* (as opposed to *semantic*) analysis. The operator can be lifted to describe whether two *side-effect sets* are non-interfering:

$$\begin{aligned} \# &: \mathit{Side\text{-}Effect} \times \mathit{Side\text{-}Effect} \to \{0, 1\} \\ \# &: \mathscr{P}(\mathit{Side\text{-}Effect}) \times \mathscr{P}(\mathit{Side\text{-}Effect}) \to \{0, 1\} \end{aligned}$$

Two side effects are *non-interfering* if both their operations are *reads* or they act on *different channels*. The only exception to the latter is if both side effects act on file-I/O channels, in which case they are non-interfering if the file references are disjoint. This produces the following definition, $\forall (o_1, c_1), (o_2, c_2) \in \textit{Side-Effect}$:

$$\begin{aligned}(o_1, c_1) \mathrel{\#} (o_2, c_2) \leftrightarrow{}& (o_1 = \mathrm{R} \wedge o_2 = \mathrm{R}) \\ &\vee \neg(c_1 = \text{file\_ref}\,(i_1, g_1) \wedge c_1 = \text{file\_ref}\,(i_2, g_2)) \wedge (c_1 \neq c_2) \\ &\vee \quad (c_1 = \text{file\_ref}\,(i_1, g_1) \wedge c_1 = \text{file\_ref}\,(i_2, g_2)) \wedge (i_1 \neq i_2 \wedge g_1 = g_2)\end{aligned}$$

Two side-effect sets f_1, f_2 are non-interfering if *all* side effects in f_1 are non-interfering with *all* side effects in f_2:

$$f_1 \mathrel{\#} f_2 \leftrightarrow (\forall \sigma_1 \in f_1, \sigma_2 \in f_2 \cdot \sigma_1 \mathrel{\#} \sigma_2)$$

A side-effect system can be implemented using the *abstract analysis framework*. The monadic return type stores the expression's side-effect set f and the accumulator stores the context Γ^{se}. A representative subset of typing rules are described below, with a full list provided in Appendix B.

In general, the side effects produced by an expression are the union of its sub-expression's side effects, plus any new side effects introduced by the top-level operation. For example, if `e` produces the side effect f, then `print(e)` produces the side effects $\{\mathrm{W} : \text{console}\} \cup f$. For brevity, τ^{se} is written as τ and Γ^{se} is written as Γ:

$$\frac{}{x : \tau,\ \Gamma \vdash x : \tau, \{\mathrm{R}, \text{var}(x)\}}\ (\textit{var-read})$$

$$\frac{\Gamma \vdash e_1 : \tau_1, f_1 \qquad (x : \tau_1), \Gamma \vdash e_2 : \tau_2, f_2}{\Gamma[x : \tau_1] \vdash x = e_1;\ e_2 : \tau_2, (f_1 \cup f_2 \cup \{(\mathrm{W}, \text{var}(x))\})}\ (\textit{var-assign})$$

$$\frac{\Gamma \vdash e : \textit{string}, f}{\Gamma \vdash \text{print}(e) : \textit{unit}, f \cup \{(\mathrm{W}, \text{console})\}}\ (\textit{print}) \qquad \frac{\Gamma \vdash n : \textit{int}, \{\}}{\Gamma \vdash \text{input}(n) : \textit{string}, \{(\mathrm{W}, \text{console})\}}\ (\textit{input-1}^{3})$$

$$\frac{\Gamma \vdash e : \textit{file\_ref}(i, g), f \qquad \Gamma \vdash n : \textit{int}, \{\}}{\Gamma \vdash \text{read}(e, n) : \textit{string}, f \cup \{(\mathrm{R}, \text{file}(i, g))\}}\ (\textit{file-read-1}^{3})$$

$$\frac{\Gamma \vdash e_1 : \textit{file\_ref}(i, g), f_1 \qquad \Gamma \vdash e_2 : \textit{string}, f_2}{\Gamma \vdash \text{write}(e_1, e_2) : \textit{unit}, (f_1 \cup f_2 \cup \{(\mathrm{W}, \text{file}(i, g))\})}\ (\textit{file-write})$$

The system infers all side effects which can be *potentially* be produced by the expression. Since this analysis is *syntactic*, we add side effects from *all* possible paths in the presence of branches.

$$\frac{\Gamma \vdash e_1 : \textit{bool}, f_1 \qquad \Gamma \vdash e_2 : \textit{unit}, f_2 \qquad \Gamma \vdash e_3 : \textit{unit}, f_3}{\Gamma \vdash \text{if } e_1 \{\, e_2 \,\} \text{ else } \{\, e_3 \,\} : \textit{unit}, (f_1 \cup f_2 \cup f_3)}\ (\textit{if})$$

[3] The extra argument n is required for type-cost analysis (section 3.5.1).

We recall from section 2.4 that local-variable side effects are removed from the side-effect set of scopes. The following example describes how this is applied to a *code block*. However, the same rule is applied to all other scopes as well (for example in *if-statements* and *functions*), but are omitted from this section for brevity. We use the notation $f \setminus \{x, y, \ldots z\}$ to represent the removal of variable side effects affecting variables $x, y, \ldots, z$ from the side-effect set f:

$$\frac{\Gamma \vdash e : \mathit{unit}, f}{\Gamma \vdash \{e\}_{\{x,y,\ldots,z\}} : \mathit{unit}, f \setminus \{x, y, \ldots, z\}}\ (\mathit{block})$$

The function typing rules follow directly from the definitions of latent effects (section 2.3.2):

$$\frac{x_1 : \tau_1,\ \ldots, x_n : \tau_n, \Gamma \vdash e : \tau, f \qquad (g : \tau_1 * \cdots * \tau_n \xrightarrow{f} \tau), \Gamma \vdash e' : \tau', f'}{\Gamma \vdash (\text{def } g(x_1 : \tau_1,\ \ldots, x_n : \tau_n)\ \tau\ \{\, e\, \};\ e') : \tau', f'}\ (\mathit{def\text{-}func})$$

$$\frac{\Gamma(g) = \tau_1 * \cdots * \tau_n \xrightarrow{f} \tau \qquad \Gamma \vdash e_1 : \tau_1, f_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n, f_n}{\Gamma \vdash g(e_1, \ldots, e_n) : \tau, (f \cup f_1 \cup \cdots \cup f_n)}\ (\mathit{apply\text{-}func})$$

Section 2.1.3 provides full examples of side-effect inference and cost analysis. The following sections will reiterate these examples, in light of details describing how the analysis is performed.

```
// No side effects
func f(a int) int {
  return a + 1
}

// Side effects = { Write : Console } ← (x, y local)
func g(x int, y int) int {

  // Side effects = { Read : Var(x), Write : Console } ← (z local)
  {
    z := f(x)
    print(z)
  }

  // Side effects = { Read : Var(x), Write : Var(y) }
  {
    y = x + 10
  }

  // These side effects are non-interfering
  // So it is safe to parallelise these blocks
}
```

Key Takeaway. *We can infer the side effects produced by expressions with a side-effect system. File-reference aliasing tells us whether two file references are* ***disjoint****, which is used by the* ***non-interfering operator*** *# to determine if two side effects are non-interfering. If the side-effect sets produced by adjacent code blocks are* ***non-interfering*** *then the blocks are* ***safe*** *to parallelise.*

3.5 Cost Analysis

This section aims to implement the *type-cost system* and *runtime-cost systems* introduced in the preparation chapter. It is cumbersome for the programmer to annotate sizes for all types in the program, so we implement a *powerful type-cost inference algorithm* to minimise programmer overhead.

3.5.1 Type-Cost Analysis

Binary cost-bound operations can be applied to the cost bounds $c_1 = \langle l_1, u_1 \rangle$, $c_2 = \langle l_2, u_2 \rangle$ as follows:

$$\begin{aligned} c_1 + c_2 &\triangleq \langle l_1 + l_2, u_1 + u_2 \rangle \\ c_1 \cdot c_2 &\triangleq \langle l_1 \cdot l_2, u_1 \cdot u_2 \rangle \\ c_1 \cup c_2 &\triangleq \langle \min(l_1, l_2), \max(u_1, u_2) \rangle \\ c_1 - c_2 &\triangleq \langle l_1 - u_2, u_1 - l_2 \rangle \\ c_1 \dot{-} c_2 &\triangleq \langle l_1 - l_2, u_1 - u_2 \rangle \end{aligned}$$

Appendix C provides more detail on how these definitions are derived, and describes how operations can be applied to individual costs (e.g. $l_1 + l_2$). We recall from section 2.1.2 that Kautuka only supports non-negative numbers, which simplifies our definitions for subtraction and multiplication. The union operator $c_1 \cup c_2$ produces the tightest bound which encapsulates both c_1 and c_2, and the change operator $c_1 \dot{-} c_2$ returns the cost which can be added back to c_2 to obtain c_1.

Our primitives include *integers* and *strings*, whose type costs are defined as their numerical values and lengths respectively. For brevity, τ^{cost} is written as τ and Γ^{cost} is written as Γ. The remaining typing rules are listed in appendix C.

$$\frac{}{\Gamma \vdash n : \mathit{int}\langle n, n \rangle}\ (\mathit{int}) \qquad \frac{\text{len}(s) = n}{\Gamma \vdash s : \mathit{string}\langle n, n \rangle}\ (\mathit{string})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \quad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1 + e_2 : \mathit{int}(c_1 + c_2)}\ (\mathit{add}) \qquad \frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \quad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1 - e_2 : \mathit{int}(c_1 - c_2)}\ (\mathit{sub})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \quad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1 * e_2 : \mathit{int}(c_1 \cdot c_2)}\ (\mathit{mult}) \qquad \frac{\Gamma \vdash e_1 : \mathit{string}(c_1) \quad \Gamma \vdash e_2 : \mathit{string}(c_2)}{\Gamma \vdash e_1 + e_2 : \mathit{string}(c_1 + c_2)}\ (\mathit{concat})$$

However, it is impossible to statically estimate the size of user-I/O inputs without user intervention: programmers are required to supply an upper bound (and optionally a lower bound, 0 by default) as an argument to the I/O functions. For example `input(n)` returns a value upper bounded by n and lower bounded by 0, whereas `input(n1, n2)` returns a value with size bounds $\langle n_1, n_2 \rangle$.

$$\frac{\Gamma \vdash n : \mathit{int}\langle n, n \rangle}{\Gamma \vdash \text{input}(n) : \mathit{int}\langle 0, n \rangle}\ (\mathit{input\text{-}1}) \qquad \frac{\Gamma \vdash n_1 : \mathit{int}\langle n_1, n_1 \rangle \quad \Gamma \vdash n_2 : \mathit{int}\langle n_2, n_2 \rangle \quad n_1 \leq n_2}{\Gamma \vdash \text{input}(n_1, n_2) : \mathit{int}\langle n_1, n_2 \rangle}\ (\mathit{input\text{-}2})$$

$$\frac{\Gamma \vdash f : \mathit{file\_ref}(i, g) \qquad \Gamma \vdash n : \mathit{int}\langle n, n\rangle}{\Gamma \vdash \mathrm{read}(f, n) : \mathit{string}\langle 0, n\rangle}\ (\mathit{read\text{-}1})$$

$$\frac{\Gamma \vdash f : \mathit{file\_ref}(i, g) \qquad \Gamma \vdash n_1 : \mathit{int}\langle n_1, n_1\rangle \qquad \Gamma \vdash n_2 : \mathit{int}\langle n_2, n_2\rangle \qquad n_1 \leq n_2}{\Gamma \vdash \mathrm{read}(f, n_1, n_2) : \mathit{string}\langle n_1, n_2\rangle}\ (\mathit{read\text{-}2})$$

The rest of this section details the novel typing rules and algorithms I developed to infer cost-types in the presence of imperative control flow and functions.

An *if-else* statement is a *unit-type* expression, and so its typing rule depicts that it will always return *unit*:

$$\frac{\Gamma \vdash e_1 : \mathit{bool} \qquad \Gamma \vdash e_2 : \mathit{unit} \qquad \Gamma \vdash e_3 : \mathit{unit}}{\Gamma \vdash \mathrm{if}\ e_1\ \{\, e_2 \,\}\ \mathrm{else}\ \{\, e_3 \,\} : \mathit{unit}}\ (\mathit{if\text{-}else})$$

However, this typing rule does not change the typing context accordingly. The example below shows a case where the context may differ depending on the path taken:

```
x := 0

if condition {
  x += 1
} else {
  x += 2
}
```

The type-cost context now either contains $x \mapsto \langle 1, 1\rangle$ or $x \mapsto \langle 2, 2\rangle$, depending on which path was taken. The natural solution is for the context to encapsulate all possible paths taken: $x \mapsto \langle 1, 1\rangle \cup \langle 2, 2\rangle = \langle 1, 2\rangle$.

To implement this, we start by taking the contexts at the end of both branches (Γ_{if} and Γ_{else}) and remove local variables declared within the branches. This ensures that both Γ_{if} and Γ_{else} have the same size (that is, contain the same variables in the same positions); the only variables that can be added to these contexts are local variables, which are subsequently removed. Hence, both contexts have corresponding mappings (e.g. $y \mapsto \tau(c_1)$, $y \mapsto \tau(c_2)$), with the same variables and base types but potentially differing cost bounds. We lift the *binary cost-bound operator* $\cup$ to the context level, applying the operator to all corresponding cost bounds in the contexts. For example if $\Gamma_{\text{if}} = \{x : \mathit{int}\langle 1, 1\rangle\}$ and $\Gamma_{\text{else}} = \{x : \mathit{int}\langle 2, 2\rangle\}$ then $\Gamma_{\text{end}} = \Gamma_{\text{if}} \cup \Gamma_{\text{else}} = \{x : \mathit{int}\langle 1, 2\rangle\}$. Γ_{end} represents the context at the end of the *if-statement.*

For-loops face similar problems, the typing rule for *for-loops* (excluding context changes) is:

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \qquad \Gamma \vdash e_2 : \mathit{int}(c_2) \qquad \Gamma \vdash e_3 : \mathit{unit}}{\Gamma \vdash \mathrm{for}\ x := e_1\,;\ x < e_2\,;\ x{+}{+}\ \{\, e_3 \,\} : \mathit{unit}}\ (\mathit{for\text{-}loop})$$

Consider a general *for-loop*: `for i := e1; i < e2; i++ { e3 }`, with expression type costs $e_1 : \mathit{int}(c_1)$, $e_2 : \mathit{int}(c_2)$, $e_3 : \mathit{unit}$ (assuming that $e_2 \geq e_1$). The number of iterations n is described with the cost bound $n = c_2 - c_1$. However, it is statically undecidable to determine n's exact value as it can parameterised on unknown input sizes.

The *for-loop* starts with the context Γ_{start}. We calculate the context after a single loop iteration Γ_{iter}, where the iterator variable i has type cost $\mathit{int}(c_1 \cup c_2)$ to encapsulate both the upper and lower values of its range. Note that for Kautuka, this is sufficient to capture behaviour for all values of i within this bound (appendix C). The change to the context $\Gamma_{\text{iter}} \dot{-} \Gamma_{\text{start}}$ is calculated by lifting the *change operator* $\dot{-}$ from cost bounds to contexts. Hence, the change after n iterations is calculated by $n \cdot (\Gamma_{\text{iter}} \dot{-} \Gamma_{\text{start}})$, where $c \cdot \Gamma$ is defined as multiplying all cost bounds in Γ by the cost bound c.

However, this only represents an *approximation* of the context change, struggling to handle cases where variables blow up exponentially (e.g. `for i := e1; i < e2; i++ { x := x * x }`). Harrison [17] introduces techniques to handle a subset of these cases, however this analysis is outside the scope of the project. While this approximation limits the accuracy of our predictions, the majority of programs (especially those focused on I/O) do not exhibit exponential looping behaviour.

The final context Γ_{end} is defined by adding the overall context change to the initial context: $\Gamma_{\text{end}} = \mathit{int}(c_2 - c_1) \cdot (\Gamma_{\text{iter}} \dot{-} \Gamma_{\text{start}}) + \Gamma_{\text{start}}$. A full example is provided on the following code snippet:

```
 ...  // Ğ= { total: int<0,0>, x: int<0,5> }

for i := x; i < x + 5; i++ {
  total += i
}
```

For the single iteration, i has the type cost $\mathit{int}(c_1 \cup c_2) = \mathit{int}(\langle 0,5\rangle \cup \langle 5,10\rangle) = \mathit{int}\langle 0,10\rangle$. This produces the context change after a single iteration: $\Gamma_{\text{iter}} = \{\mathit{total} : \mathit{int}\langle 0,10\rangle, x : \mathit{int}\langle 0,5\rangle\}$. Using the above formula, the final context value is:

$$\begin{aligned}
\Gamma_{\text{end}} &= \mathit{int}(c_2 - c_1) \cdot (\Gamma_{\text{iter}} \dot{-} \Gamma_{\text{start}}) + \Gamma_{\text{start}} \\
&= \mathit{int}(\langle 5,10\rangle - \langle 0,5\rangle) \cdot (\{\mathit{total} : \mathit{int}\langle 0,10\rangle, x : \mathit{int}\langle 0,5\rangle\} \dot{-} \{\mathit{total} : \mathit{int}\langle 0,0\rangle, x : \mathit{int}\langle 0,5\rangle\}) \\
&\quad + \{\mathit{total} : \mathit{int}\langle 0,0\rangle, x : \mathit{int}\langle 0,5\rangle\} \\
&= \mathit{int}\langle 0,10\rangle \cdot \{\mathit{total} : \mathit{int}\langle 0,10\rangle, x : \mathit{int}\langle 0,0\rangle\} + \{\mathit{total} : \mathit{int}\langle 0,0\rangle, x : \mathit{int}\langle 0,5\rangle\} \\
&= \{\mathit{total} : \mathit{int}\langle 0,100\rangle, x : \mathit{int}\langle 0,0\rangle\} + \{\mathit{total} : \mathit{int}\langle 0,0\rangle, x : \mathit{int}\langle 0,5\rangle\} \\
&= \{\mathit{total} : \mathit{int}\langle 0,100\rangle, x : \mathit{int}\langle 0,5\rangle\}
\end{aligned}$$

Which statically tells us that *total* must be between 0 and 100 at the end of this loop.

Latent effect rules for type-cost systems (section 2.5.2) describes how each function (returning a sized type) generates a mapping from its input sizes to its output size. When calling these functions, we look up this mapping and pass in the input sizes.

(*def-func-2*):

$$\frac{x_1 : \tau_1(c_1), \ldots, x_n : \tau_n(c_n), \Gamma \vdash e : \tau_{\text{sized}}(c) \qquad (g : \tau_1^{\text{base}} * \cdots * \tau_n^{\text{base}} \xrightarrow{g_{size}(c_1,\ldots,c_n)=c} \tau^{\text{base}}), \Gamma \vdash e' : \tau'}{\Gamma \vdash (\text{def } g(x_1 : \tau_1^{\text{base}}, \ldots, x_n : \tau_n^{\text{base}})\ \tau^{\text{base}}\ \{\, e \,\};\ e') : \tau'}$$

Note that if expression e in *(def-func-2)* returns multiple values[4] of sizes $c', c'', \ldots, c^{(n)}$, then the size of e is defined as $c = c' \cup c'' \cup \cdots \cup c^{(n)}$.

[4] e must contain at least one return statement as the return statement of the function is *sized*, and hence *non-unit*.

(*apply-func-2*):

$$\frac{\Gamma \vdash g : (\tau_1^{\text{base}} * \cdots * \tau_n^{\text{base}}) \xrightarrow{g_{size}(c_1,\ldots,c_n)=c} \tau^{\text{base}} \qquad \Gamma \vdash e_1 : \tau_1(c_1) \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n(c_n)}{\Gamma \vdash g(e_1,\ldots,e_n) : \tau_{\text{sized}}(g_{size}(c_1,\ldots,c_n))}$$

We recall the following example of type-cost inference from section 2.1.3, noting that exact values are provided as opposed to bounds.

```
// f_size(a) = a + 1 (maps input size to output size)
func f(a int) int {
  return a + 1
}

func g(x int, y int) int {

  {
    z := f(x)   // Size of z = f_size(x) = x + 1
    print(z)
  }

  {
    y = x + 10  // Size of y = x + 10
  }

}
```

3.5.2 Runtime-Cost Analysis

Static profiling estimates the runtime of each instruction, as a function of their input sizes. By measuring instruction runtimes for various input sizes, we produce accurate *instruction runtime estimates*. A list of instruction runtime estimates (produced by my machine) are provided in appendix D.

The runtime-cost system uses these estimates to generate *expression runtime estimates*. In general, the runtime of an expression is the sum of its sub-expression runtimes plus the top-level instruction runtime. For example `e_1 + e_2` takes $(r_1 + r_2 + \text{ADD}(c_1, c_2))\mu s$ where $e_1 : \mathit{int}(c_1)$, $e_2 : \mathit{int}(c_2)$ and the runtimes of e_1 and e_2 are r_1 and r_2. A representative subset of these typing rules are provided below, with the remainder listed in appendix D.

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1), r_1 \qquad \Gamma \vdash e_2 : \mathit{int}(c_2), r_2}{\Gamma \vdash e_1 + e_2 : \mathit{int}(c_1 + c_2), r_1 + r_2 + \text{ADD}(c_1, c_2)}\,(\mathit{add})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1), r_1 \qquad \Gamma \vdash e_2 : \mathit{int}(c_2), r_2}{\Gamma \vdash e_1 * e_2 : \mathit{int}(c_1 \cdot c_2), r_1 + r_2 + \text{MULT}(c_1, c_2)}\,(\mathit{mult})$$

If-statements compute the maximum runtime of both branches using the $\cup$ operator. *For-loops* calculate the runtime of all iterations by multiplying the runtime of a single loop iteration (plus a comparison and increment operation) by the number of iterations. IF() and FOR_LOOP() refer to extra costs incurred by initialising the control structures.

$$\frac{\Gamma \vdash e_1 : \mathit{bool}, r_1 \qquad \Gamma \vdash e_2 : \mathit{unit}, r_2 \qquad \Gamma \vdash e_3 : \mathit{unit}, r_3}{\Gamma \vdash \text{if } e_1 \; \{ \, e_2 \, \} \text{ else } \{ \, e_3 \, \} : \mathit{unit}, r_1 + (r_2 \cup r_3) + \text{IF}()} \; (\textit{if-else})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1), r_1 \qquad \Gamma \vdash e_2 : \mathit{int}(c_2), r_2 \qquad \Gamma \vdash e_3 : \mathit{unit}, r_3}{\begin{array}{c}\Gamma \vdash \text{for } x := e_1 \, ; \, x < e_2 \, ; \, x{+}{+} \; \{ \, e_3 \, \} : \mathit{unit}, \\ r_1 + r_2 + (c_2 - c_1) \cdot (r_3 + \text{LE}(c_2 - c_1, c_2) + \text{INC}(c_2 - c_1)) + \text{FOR\_LOOP}()\end{array}} \; (\textit{for-loop})$$

The function typing rules below follow similarly from those in type-cost analysis.

(*def-func-1*):

$$\frac{x_1 : \tau_1(c_1), \, \ldots \, , x_n : \tau_n(c_n), \Gamma \vdash e : \tau_{\text{unsized}}, r \qquad (g : \tau_1^{\text{base}} * \cdots * \tau_n^{\text{base}} \xrightarrow{g_{runtime}(c_1,\ldots,c_n)=r} \tau^{\text{base}}), \Gamma \vdash e' : \tau', r'}{\Gamma \vdash (\text{def } g(x_1 : \tau_1^{\text{base}}, \, \ldots \, , x_n : \tau_n^{\text{base}}) \; \tau^{\text{base}} \; \{ \, e \, \}; \; e') : \tau', r'}$$

(*apply-func-1*):

$$\frac{\Gamma \vdash g : (\tau_1^{\text{base}} * \cdots * \tau_n^{\text{base}}) \xrightarrow{g_{runtime}(c_1,\ldots,c_n)=r} \tau^{\text{base}} \qquad \Gamma \vdash e_1 : \tau_1(c_1), r_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n(c_n), r_n}{\Gamma \vdash g(e_1, \ldots, e_n) : \tau_{\text{unsized}}, (r_1 + \cdots + r_n + g_{runtime}(c_1 + \cdots + c_n) + \text{FUNC\_CALL}())}$$

With this knowledge, we review the example of runtime-cost analysis presented in section 2.1.3. The instruction runtimes can be summarised with $\text{ADD}(a, b) = 2a + 2b$ and $\text{PRINT}(a) = 100a$, where all other instructions take no time.

```
// f_runtime(a) = 2a + 2 (maps input size to runtime)
func f(a int) int {
  return a + 1
}

func g(x int, y int) int {

  // Runtime expression = 102x + 102
  {
    z := f(x)   // Runtime = f_runtime(x) = 2x + 2
    print(z)    // Runtime = 100 * f_size(z) = 100(x + 1) = 100x + 100
  }

  // Runtime expression = 2x + 20
  {
    y = x + 10  // Runtime = 2x + 2(10) = 2x + 20
  }

}
```

Key Takeaway. *Type-cost inference annotates each expression with their type cost. Runtime-cost inference builds on top of this analysis, annotating each code block with their estimated runtime. The algorithms presented in this section overcome challenges posed by implementing cost analysis in the context of imperative languages.*

3.6 Parallelisation

Sequential, non-interfering code blocks can be re-ordered and parallelised in multiple ways, in a process known as *clustering*. For example, if sequential blocks `A; B; C` all contain non-interfering side effects, then they can be re-ordered (e.g. `C; A; B`) and then parallelised (e.g. `(C | A); B` and `C | A | B`) in any way. The notation `X; Y` represents the sequential execution of blocks `X` and `Y`, and `X | Y` represents their parallel execution. However, this process becomes more complex when block side effects do interfere, especially since interference is non-transitive. Section 3.6.1 presents an algorithm to cluster blocks, optimising their performance under certain assumptions.

Section 3.6.2 describes how to translate clustered blocks into Go, introducing `sync` instructions to block program execution until all threads have terminated.

3.6.1 Re-order and Parallelise Code Blocks

It is impossible to evaluate *runtime-cost expressions* at compile time, as they can be parameterised on unknown input sizes. This means we cannot infer any information about code-block runtimes without analysing all function call sites. So during clustering, we assume all code-block runtimes to be the same. Whilst this assumption may appear unfavourable, this introduces the highest degree of parallelisation into clustering — providing near-optimal performance in the majority of cases.

A *parallelisation group* consists of blocks executed in parallel (`A | B | ...`). By interleaving parallelisation groups and sequential execution (e.g., `((A | B); C) | D`), we produce *multi-layer parallelisation groups*. However, runtime-cost expressions are evaluated at each sequential layer, so the number of evaluations scales $\mathcal{O}(n)$ with the number of blocks. Whilst runtime-cost expressions are cheap, if evaluated multiple times in the same cluster they becomes non-negligible. Hence, we constrain ourselves to only single-level parallelisation groups, consisting of sequentially executed parallelisation groups (e.g. `(A | B); C; (D | E)`), so that the number of evaluations instead scales $\mathcal{O}(1)$ with the number of blocks.

Our clustering algorithm is performed under these assumptions, which lets us approximate a cluster's runtimes as *the number of parallelisable groups* it contains. For example `(A | B); C; (D | E)` contains three sequentially-executed parallelisation groups, resulting in the approximate runtime: 3.

If two blocks have *non-interfering* side effects, they can be re-ordered and parallelised. Choosing the **optimal** clustering (under our assumptions) is difficult as there are an **exponential** number of valid allocations. Picking the allocation with the "most parallelisation" is challenging, as placing a block into a parallelisation group limits which other blocks can be placed into that group. This initially looks like a *dynamic programming* problem, however I devised a *greedy algorithm* which also produces an optimal solution by exploiting properties of non-interference. A clustering example can be seen below:

```
...
{ x = 0 } // Block A   { W : var(x) }
{ y = 0 } // Block B   { W : var(y) }
{ x = 0 } // Block C   { W : var(x) }
{ z = 0 } // Block D   { W : var(z) }
{ z = 0 } // Block E   { W : var(z) }
```

Figure 3.2: An example of sequential blocks with interfering side effects.

Naive clustering: `(A | B); (C | D); E`, runtime = 3

Optimal clustering: `(A | B | D); (C | E)`, runtime = 2

A *block list* contains a list of sequential blocks paired with their side-effect sets. From the example in fig. 3.2, we derive the block list:

```
[(A,{W:var(x)}),(B,{W:var(y)}),(C,{W:var(x)}),(D,{W:var(z)}),(E,{W:var(z)})]
```

The clustering algorithm works by passing over the block list multiple times. Each pass initialises an empty parallelisation group and an empty accumulator side-effect set f. As we iterate over the list, if a block's side effects are non-interfering with f, we extract the block from the list and place it into the parallelisation group. We then add its side effects to f (no matter if it was extracted or not). At the end of the list, we add the parallelisation group to a *parallelisation list*. Once the block list is empty, the program terminates. The parallelisation list now contains an optimal clustering of parallelisation groups, in the form:

```
par_list = [
             [A, B, D],
             [C, E]
           ]
```

Appendix E presents a proof sketch for why this algorithm is *optimal*.

Algorithm 1: Cluster Blocks into Parallelisable Groups

```
Data: block_list
Result: par_list
par_list ← [];
while not block_list.is_empty() do
    $f$ ← {};
    par_group ← [];
    for $A, f'$ in block_list do
        if $f' \# f$ then
            par_group.append($A$);
        end
        $f \leftarrow f \cup f'$;
    end
    par_list.append(par_group);
end
```

3.6.2 Translate to Parallelised Go Code

The main difficultly when translating the AST into Go is the treatment of parallelised blocks. We recall that parallelised blocks compile to both their sequential and parallel forms; the form executed depends on the evaluation of *runtime-cost expressions*. Runtime-cost expression can be transliterated from theory into code, Kautuka contains corresponding variables and operations to those used in the dependent cost calculus (appendix C).

The previous section outlines the algorithm which clusters blocks into parallelisation groups, for example producing the parallelisation list:

```
par_list = [
             [A, B, D],
             [C, E]
           ]
```

The *sequential form* is represented as `A; B; D; C; E` (or equivalently `A; B; C; D; E`) and *parallelised form* as `(A | B | D); (D | E)`. The latter consists of two parallelisation groups (`A, B, D` and `C, E`).

```
// Parallelisation group 1
{
  go func() {
    A
  }()

  go func() {
    B
  }()

  go func() {
    D
  }()
}

// Parallelisation group 2
{
  go func() {
    C
  }()

  go func() {
    E
  }()
}
```

However, the above code does nothing to prevent program execution from continuing after *parallelisation group 1*, as goroutine threads are run in the background. Hence, blocks `C` and `E` could be executed alongside blocks `A, B, C`, producing a *race*. To enforce sequential execution of parallelised groups, execution is *blocked* at the end of a group until all goroutines have terminated. This is achieved using Go's `sync` package, specifically the `WaitGroup` object: an asynchronous counter object containing `Add`, `Wait`, and `Done` methods. To parallelise n blocks, we `Add` the integer n to the `WaitGroup` counter. At the end of each block, the `Done` method is called to decrease the counter by one. The `Wait` method is placed at the end of the *parallelisable group*: it blocks execution until the counter is zero (all goroutines have terminated). The following example shows the parallelisation of code blocks `A` and `B` (with runtime costs r_a and r_b, and a constant parallelisation cost r_p).

```
// if sequential_runtime < parallel_runtime + parallelisation_costs
if r_a + r_b < min(r_a, r_b) + r_p {
  // Sequential execution
  A;
  B;
} else {

  // Parallel execution
  var wg sync.WaitGroup // Initialise async counter
  wg.Add(2) // Set counter to 2: 2 threads left to complete

  go func() { // Spawn goroutine
    A; // Runs code block
    wg.Done() // Marks thread complete
  }() // Invoke goroutine

  go func() {
    B;
    wg.Done()
  }()

  wg.Wait() // Wait until counter reaches 0
            // (all threads have termined and we can continue)
}
```

Key Takeaway. *Adjacent code blocks are parallelised* ***optimally*** *using a greedy clustering algorithm (section 3.6.1). Parallelisation is implemented into Go using goroutines, using* ***WaitGroups*** *to block program execution until all threads in a parallelised group have terminated (section 3.6.2).*

3.7 Summary

The compiler's implementation builds upon the effect systems presented in the preparation chapter. Section 3.4 presents a concrete implementation of side-effect tracking in Kautuka, utilising file-reference aliasing and introducing a *non-interference* operator to describe whether parallelisation is *safe*. Section 3.5 implements cost analysis, with a powerful inference algorithm to estimate expression runtimes. This analysis guides parallelisation in section 3.6 when translating the AST into Go. However, the author wishes to reiterate that the majority of this work is *language agnostic*, generalising well to other imperative programming languages.

Chapter 4

Evaluation

Both the automatic parallelisation compiler and Kautuka's language design **exceed the project's core success criteria**. In section 4.1, I demonstrate how the project achieves all core criteria set out in the project proposal (appendix F). I then evaluate the project beyond the original success criteria using my three research questions (listed below) as a basis. Sections 4.2 to 4.4 show that all three investigations yield **positive results**.

Research questions:

1. Transfer automatic-parallelisation theory from functional to imperative languages
2. Investigate the performance of task-based parallelisation on I/O-bound programs
3. Investigate real-world applications of the compiler

4.1 Review of Project Requirements

Reviewing the feature requirements outlined in the preparation chapter (section 2.7.1), I implemented all *Must-have* core deliverables and the majority of *Should-have* and *Could-have* extensions. Evaluating the project against the original success criteria from the project proposal (appendix F) yields:

Design a full-featured imperative language Kautuka with a compiler to Go I realised that creating a *full-featured* programming language was not appropriate for a project of this size. So I constrained the project's scope to a *mid-featured* language, with key features listed in the MoSCoW analysis of section 2.7.1. I completed all core and extension features from this table, excluding the *Could-have* feature: recursion.

Implement side-effect tracking The compiler successfully performs side-effect tracking with all proposed extensions: file I/O (with aliasing analysis) and console I/O (section 3.4).

Implement cost analysis The compiler successfully implements type-cost analysis and runtime-cost analysis. I extended type-cost analysis with a powerful inference algorithm to minimise programmer effort. Section 4.2 shows that runtime-cost estimates are sufficiently accurate to guide parallelisation.

Parallelise code based upon this analysis Section 3.6 describes how side-effect tracking and cost analysis influence parallelisation. The following success criteria analyses the efficacy of our approach.

Ensure that the produced parallel code is both *safe* and *efficient* I provide empirical evidence for *safety* through testing, performing a total of **48 tests** (section 4.2). To show that parallel code is *efficient*, I benchmarked Kautuka against the equivalent sequential Go code. Section 4.3 shows that Kautuka's performance *exceeds* that of sequential Go for I/O-bound programs by an average of 35%.

4.2 Functional to Imperative Theory

The core of high-level automatic parallelisation lies in side-effect tracking and cost analysis. While such techniques have been previously explored in the context of functional languages, implementing this theory into an imperative language compiler poses new challenges. This section evaluates the success of my solutions to these problems — defined here as preserving the *safety* of side-effect tracking and *accuracy* of cost analysis.

I provide empirical evidence for side-effect tracking *safety* through testing. I validate the correctness of each typing rule implementation with *unit tests*, writing **38 tests** to cover all top-level expression constructions. To verify that safety extrapolates to more complex expressions, I implemented **10 component tests** — testing edge cases between typing rule interactions. I translated ten non-trivial Go programs into Kautuka by hand, to produce a test corpus of Kautuka programs. These programs form the basis of my component testing, which validates the correctness of side-effect sets produced by each code block. To further substantiate claims of safety, I compared Kautuka's execution behaviour against the original Go code (repeating experiments to account for non-determinism). If these behaviours are identical, then I conclude that parallel code derived from Kautuka is *race-free.*

Results All 38 unit tests and 10 component tests passed, and Kautuka's execution behaviour was identical to Go. These are promising results, suggesting that side-effect tracking is *safe* and the produced parallel Go code is *race-free.* However, it is impossible to construct a program corpus covering all edge cases; a way to guarantee safety would be through a formal proof.

The purpose of cost analysis (in this project) is to estimate program runtime. It is sufficient in our case to predict runtimes of the correct *order of magnitude.* Hence, cost analysis is *successful* if our predictions are within one order of magnitude (base 10) of the actual runtime. We use the corpus of Kautuka programs, mentioned previously, to compare *predicted* and *actual runtimes.*

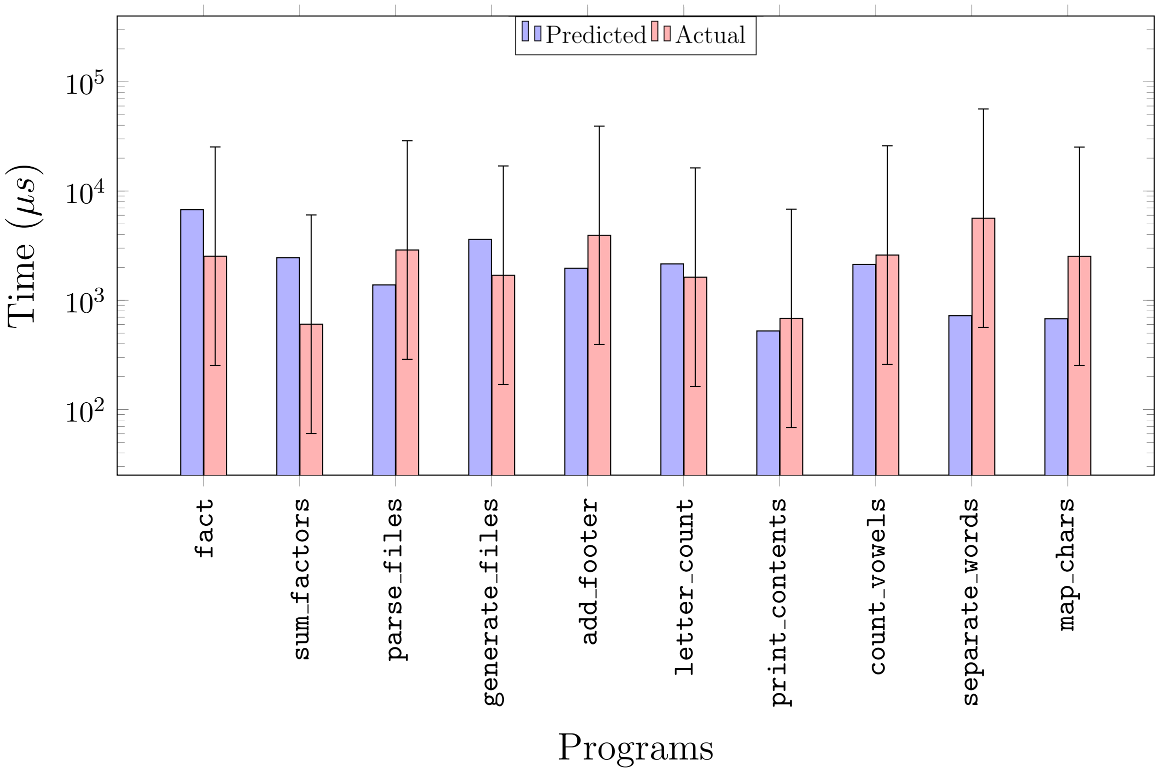

Figure 4.1: Comparing *predicted* and *actual runtimes* of Kautuka with 1000 trials — error bars (unusually) represent one order of magnitude above and below the mean *actual runtime.*

Measurements were collected on a Dell XPS 15 7590 with the following specifications:

Processor Intel Core i7-9750H CPU @ 2.60GHz × 12

Memory 16 GB 2667 MHz DDR4 RAM

OS Zorin OS 16.2

Go Version go1.19 linux/amd64

The primary source of non-determinism was file-I/O caching. To minimise this effect, we space experiments across set time intervals, separated by redundant operations to clear the cache. These measurements were automated with a bash script, using the Unix `time` command.

Results In all experiments, our predicted runtime was within an order of magnitude of the true runtime (fig. 4.1). One may argue that classifying success as "within one order of magnitude" may be too permissive. However, section 4.3 proves this bound to be sufficiently tight for Kautuka to *consistently outperform* Go. Additionally, there are unpredictable runtime factors (such as caching and inter-process communication) which limit the accuracy of cost analysis.

The programs producing the worst estimates were: `sum_factors`, `separate_words`, and `map_chars`. After investigation, I discovered that the root cause of these inaccuracies were large blocks of non-I/O computations. One explanation for why cost analysis performs worse on these computations is that runtimes of non-I/O operation are orders of magnitude shorter than for I/O operations, measured as a matter of nanoseconds. This requires operations to be executed multiple times in a single run during static profiling, to obtain measurable runtimes. However, repeating operations in such a way makes measurements susceptible to caching effects, reducing the accuracy of static analysis and hence runtime estimates.

However, all other program estimates were accurate, concluding that my **novel** approach to cost analysis produces *successful* I/O runtime predictions. Since I/O computations dominate runtime in I/O-bound programs (the primary candidates for automatic parallelisation), I constitute this analysis to be a *success*.

4.3 Performance on I/O-bound Programs

This section investigates the performance of Kautuka on I/O-bound programs; substantiating claims that Kautuka outperforms Go. Since no existing automatic-parallelisation solutions compile to Go, it would be meaningless to compare their performances with my compiler. Runtime discrepancies would be primarily attributed to the target languages' differing performance, rather than the effectiveness of parallelisation. Thus, the most suitable comparison is with sequential Go, which entails equivalent programming effort to Kautuka (section 4.4).

Programmers using the compiler are expected to identify candidate Go files containing expensive (in our case specifically I/O) operations. I performed this methodology on the Kautuka corpus, removing files which fail to satisfy this criteria (`fact` and `sum_factors`). The following benchmarks (fig. 4.2) compare the performance of the Kautuka against sequential Go code. By their nature, all chosen Kautuka programs exhibited some degree of parallelisation during execution — I denote programs containing a combination of both sequential and parallel execution paths with †.

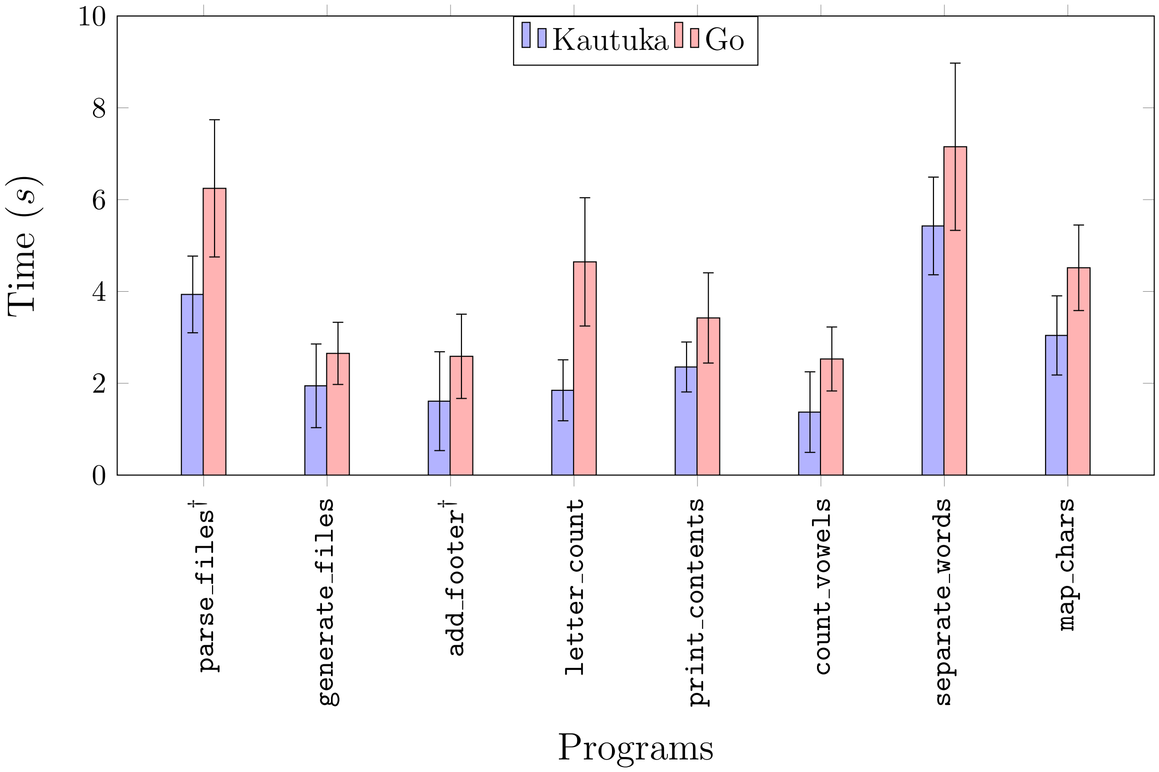

Figure 4.2: Benchmarking Kautuka against Go. Error bars representing $\pm\sigma$

Results Our benchmarks show that **Kautuka consistently outperforms sequential Go**, with performance improvements most pronounced on programs `letter_count` and `count_vowels`. These programs contain *for-each* loops iterating over file contents, so we deduce that these types of programs exhibit greater performance gains. Overall, these results demonstrate that high-level automatic parallelisation *improves* the performance of I/O-bound programs.

4.4 Real-World Compiler Applications

To justify claims that Kautuka integrates into existing workflows, we first illustrate the translation process (from an existing Go file into Kautuka). As is expected with a project of this size, Kautuka lacks some features present in Go such as error handling and file access controls — which we ignore when translating Go to Kautuka. However, there are no theoretical limitations as to why such features could not be included in the language.

Results Listing 1 presents a transliteration of Go to Kautuka with *minimal extra overhead*. Notably, the required annotations (size bounds on I/O inputs) are less strenuous to add than typical parallelisation primitives; the notion of *input sizes* is more grounded than *parallel execution behaviour*, reducing the likelihood of introducing errors. Since Kautuka's syntax is similar to that of Go, and few extra annotations are required, I conclude that Go files can be translated into Kautuka with *minimal effort*.

```go
package main

import ("fmt"; "os")

func f(x int) {
  for i := 0; i < x; i++ {
    print(i)
  }
}

func main() {

  var user_input string
  fmt.Scan(&user_input)

  user_input_len := len(user_input)

  if user_input_len > 10 {
    f(user_input_len)
  } else {
    file, _ := os.OpenFile(user_input,
                           os.O_RDWR, 0777)
    defer file.Close()

    dat, _ := os.ReadFile(file.Name())
    print(string(dat))
  }
}
```

(a) Go

```
package main

// No explicit imports required

func f(x int) {
  for i := 0; i < x; i++ {
    print(i)
  }
}

func main() {

  user_input := input(100)


  user_input_len := len(user_input)

  if user_input_len > 10 {
    f(user_input_len)
  } else {
    file := open(user_input)



    dat := read(file, 1000)
    print(dat)
  }
}
```

(b) Kautuka

Listing 1: An example I/O-bound program, hand-translated from Go (left) to Kautuka (right).

After translating candidate Go files into Kautuka, the codebase is compiled with my compilation tool. The tool automatically generates parallel Go code from Kautuka files and embeds them into the codebase (deleting Go files once no longer required).

Results Since this tool handles extra complications of a heterogeneous language repository, I conclude that the process of integrating Kautuka into existing Go workflows is *effortless*.

4.5 Summary

Kautuka exceeds all success criteria, achieving all core requirements and the majority of extensions listed in Section 2.7. Section 4.2 presents empirical evidence that my novel approach to automatic parallelisation was successful, substantiating the *safety* of side-effect tracking and *accuracy* of cost analysis when implemented in an imperative setting. Our benchmarks reveal that Kautuka outperforms Go on I/O-bound programs (section 4.3), highlighting the practicality of high-level automatic parallelisation. Section 4.4 describes the ease of translating Go files into Kautuka, and how the execution of Kautuka-containing Go codebases requires no additional effort — showcasing Kautuka's seamless integration into existing workflows.

Chapter 5

Conclusions

The project was a success. I exceeded all core success criteria and completed the majority of proposed extensions. The project's objective was to create a mid-featured, imperative programming language with a corresponding high-level automatic parallelisation compiler. I implemented both components successfully, ensuring that resultant parallel code was both *safe* and *efficient*. To the best of the author's knowledge, this is the first automatic parallelisation compiler for an imperative language which employs both side-effect tracking and cost analysis. This project highlights potential performance improvements in imperative languages, achieved through high-level automatic parallelisation.

Evaluating my project (using my three research questions as a basis) yielded positive results:

1. *Transfer automatic-parallelisation theory from functional to imperative languages*

   I faced various challenges when implementing automatic-parallelisation theory into an imperative language. To overcome these challenges, I devised **novel typing rules and algorithms**, which describe how to perform cost analysis in the presence of imperative-style functions and control flow (section 3.5). My solutions successfully preserve the *safety* of side-effect tracking and the *accuracy* of cost analysis (section 4.2).

2. *Investigate the performance of high-level parallelisation on I/O-bound programs*

   Benchmarking Kautuka against Go on a corpus of I/O-bound programs provided empirical evidence that **Kautuka outperforms Go** (section 4.3).

3. *Investigate real-world applications of the compiler*

   Powerful inferences algorithms (section 3.5) allowed me to translate hundreds of lines of Go into Kautuka with **minimal programmer overhead**. This demonstrates that Kautuka can seamlessly integrate into existing Go workflows using my compiler tooling (section 4.4).

5.1 Lessons Learnt

A significant proportion of the project's focus was on *exploration*, with the majority of time allocated to designing and implementing new algorithms, as opposed to building upon existing work. Whilst rewarding, the research required a substantial initial time investment which I failed to account for in my original timetable. Fortunately, I allocated sufficient slack time to avoid project delays. However, more careful planning would have prevented me deviating from my planned schedule.

I initially intended to implement the entire project with a *spiral development model*. However, I quickly discovered that each compiler stage was heavily dependent on its previous stages, hindering my ability to make incremental iterations. So, despite initial reservations, I used the *waterfall model* for the core project development. I originally avoided this model due to its *inflexibility*, but in reality I found this model to be very effective as all project requirements and deadlines were all fixed in advance. Writing pseudocode for all core components (before starting development) allowed me to commit to design decisions early on, without regretting them at later waterfall stages. This taught me that inflexible development models can be more effective than flexible models in projects where the scope is fixed.

Implementing the type-cost system took significantly longer than expected. The effect system's design was dependent on complex *cost* definitions (appendix C), which were challenging to implement. On reflection, it would have been beneficial to constrain the core criteria's scope as I now believe it to be ambitious for the time frame provided. Developing a full type-cost system could have been implemented as a stretch goal rather than as part of the core requirements.

5.2 Future Work

This project demonstrates that high-level automatic parallelisation[1] of imperative programs is not only feasible, but also outperforms equivalent sequential code on I/O-bound programs. The natural extension would be to integrate automatic parallelisation into an industry-standard imperative language, such as Go. Common use cases for Go include developing web servers and distributed network applications, which are inherently I/O bound. This suggests that Go is a prime candidate for high-level automatic parallelisation — with many core ideas of language having already been discussed throughout the dissertation.

Kautuka does not currently support negative numbers, however these pose no fundamental theoretical limitations. Implementing negative numbers requires us to extend the *dependent cost calculus* with *min* and *max* operators. The calculus can be enriched further to support real numbers and operations such as division and modulus. However, doing so would increase the complexity of the calculus, spawning a trade-off between expressivity and the evaluation time of *runtime-cost expressions*. Extending the calculus whilst exploring this trade-off would be an interesting direction for future work.

A more explorative research direction would be to model data-type sizes as probability distributions, rather than relying on upper and lower bounds. This presents new optimisation opportunities, such as branch prediction, not investigated during this project. Probabilistic models would generate more accurate and meaningful runtime estimates, improving the efficiency of produced parallel code.

[1] Using side-effect tracking and cost analysis.

Bibliography

[1] Sudhakar Kumar et al. "Evaluation of automatic parallelization algorithms to minimize speculative parallelism overheads: An experiment". In: *Journal of Discrete Mathematical Sciences and Cryptography* 24.5 (2021), pp. 1517–1528. DOI: 10.1080/09720529.2021.1951435. eprint: https://doi.org/10.1080/09720529.2021.1951435. URL: https://doi.org/10.1080/09720529.2021.1951435.

[2] Boris Beizer. "Analytical Techniques for the Statistical Evaluation of Program Running Time". In: *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference.* AFIPS '70 (Fall). Houston, Texas: Association for Computing Machinery, 1970, pp. 519–524. ISBN: 9781450379045. DOI: 10.1145/1478462.1478537. URL: https://doi.org/10.1145/1478462.1478537.

[3] J. M. Lucassen and D. K. Gifford. "Polymorphic Effect Systems". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 47–57. ISBN: 0897912527. DOI: 10.1145/73560.73564. URL: https://doi.org/10.1145/73560.73564.

[4] Alexandru Salcianu and Martin Rinard. "Purity and Side Effect Analysis for Java Programs". In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by Radhia Cousot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 199–215. ISBN: 978-3-540-30579-8.

[5] Mariano Sigman and Stanislas Dehaene. "Brain Mechanisms of Serial and Parallel Processing during Dual-Task Performance". In: *The Journal of neuroscience : the official journal of the Society for Neuroscience* 28 (July 2008), pp. 7585–98. DOI: 10.1523/JNEUROSCI.0948-08.2008.

[6] Jean-Pierre Talpin and Pierre Jouvelot. "Polymorphic type, region and effect inference". In: *Journal of Functional Programming* 2.3 (1992), pp. 245–271. DOI: 10.1017/S0956796800000393.

[7] Brian Reistad and David K. Gifford. "Static Dependent Costs for Estimating Execution Time". In: *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994.* Ed. by Robert R. Kessler. ACM, 1994, pp. 65–78. DOI: 10.1145/182409.182439. URL: https://doi.org/10.1145/182409.182439.

[8] Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. "Using the run-time sizes of data structures to guide parallel-thread creation". In: *LFP '94.* 1994.

[9] *dune.* URL: https://github.com/ocaml/dune (visited on 05/06/2023).

[10] *OCaml programming guidelines · ocaml tutorials.* URL: https://ocaml.org/docs/guidelines (visited on 05/06/2023).

[11] *ocamlformat.* URL: https://github.com/ocaml-ppx/ocamlformat (visited on 05/06/2023).

[12] *Lexer and parser generators (ocamllex, ocamlyacc).* URL: https://v2.ocaml.org/manual/lexyacc.html (visited on 05/11/2023).

[13] François Pottier and Yann Régis-Gianas. *Menhir reference manual.* 2016. URL: https://gallium.inria.fr/~fpottier/menhir/manual.pdf (visited on 05/11/2023).

[14] Thomas Reps. “Undecidability of context-sensitive data-dependence analysis”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22.1 (2000), pp. 162–186.

[15] Bjarne Steensgaard. “Points-to Analysis in Almost Linear Time”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 32–41. ISBN: 0897917693. DOI: 10.1145/237721.237727. URL: https://doi.org/10.1145/237721.237727.

[16] Lars Ole Andersen. “Program analysis and specialization for the C programming language”. PhD thesis. Citeseer, 1994.

[17] William H. Harrison. “Compiler analysis of the value ranges for variables”. In: *IEEE Transactions on software engineering* 3 (1977), pp. 243–250.

Appendix A

Grammar

Kautuka is a sequential imperative language defined by the following grammar. Note that this grammar consists of *expressions*, *commands*, and *structures*, and does not make the simplifying assumption to treat commands and structures as unit-type expressions. In Kautuka, semicolons are interchangeable with newline characters.

pkg	package name
fn	function name
x	variable name
n	integer
b	boolean
s	string

$\mathit{program}\ ::=$

$\quad |\ \textbf{package}\ \mathit{pkg};\ \mathit{func}_1;\ \ldots\ ;\ \mathit{func}_n;\ \textbf{func main()}\{\mathit{struct}\}$

$\mathit{func}\ ::=$

$\quad |\ \textbf{func}\ \mathit{fn}\,(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \tau\ \{\mathit{struct}\}$

$\tau\ ::=$

$\quad |\ \mathit{int}$

$\quad |\ \mathit{bool}$

$\quad |\ \mathit{string}$

$\quad |\ /\!*\ \mathit{epsilon}\ */$ Unit type

$\quad |\ \mathit{file}$ File reference

$\mathit{struct}\ ::=$ Structure

$\quad |\ \textbf{if}\ e\,\{\mathit{struct}\}$

$\quad |\ \textbf{if}\ e\,\{\mathit{struct}_1\}\ \textbf{else}\ \{\mathit{struct}_2\}$

$\quad |\ \textbf{if}\ e_1\,\{\mathit{struct}_1\}\ \textbf{else if}\ e_2\,\{\mathit{struct}_2\}\ldots\ \textbf{else}\ \{\mathit{struct}_n\}$

$\quad |\ \textbf{for}\ x := e_1;\ x < e_2;\ x{+}{+}\ \{\mathit{struct}\}$

$\quad |\ \textbf{for}\ x := \textbf{range}\ e\,\{\mathit{struct}\}$

$\quad |\ \{\mathit{struct}\}$ Code block

$\quad |\ \mathit{struct}_1;\ \mathit{struct}_2$

$\quad |\ \mathit{command}_1;\ \ldots\ ;\mathit{command}_n$

$command$ $::=$		
	$\mid$ $x := e$	Declaration (with assignment)
	$\mid$ **var** x	Declaration (without assignment)
	$\mid$ **var** $x\ \tau$	Declaration (with explicit type)
	$\mid$ $x = e$	Assignment
	$\mid$ $fn(e_1, \ldots, e_n)$	Call (user-defined) func
	$\mid$ **return** e	Return expr from func
	$\mid$ **print**(e)	Console print
	$\mid$ **write**(e_1, e_2)	File write (overwrite contents)
	$\mid$ **append**(e_1, e_2)	File append
e $::=$		
	$\mid$ $unop\ e$	
	$\mid$ $e_1\ binop\ e_2$	
	$\mid$ n	Integer
	$\mid$ b	Boolean
	$\mid$ s	String literal
	$\mid$ x	Variable read
	$\mid$ (e)	
	$\mid$ **input**(n)	Console input (lower bound)
	$\mid$ **input**(n_1, n_2)	Console input (both bounds)
	$\mid$ **open**(e)	File open (unspecified group)
	$\mid$ **open**(e, n)	File open (specified group)
	$\mid$ **read**(e, n)	File read (lower bound)
	$\mid$ **read**(e, n_1, n_2)	File read (both bounds)
$unop$ $::=$		
	$\mid$ $!$	Not
	$\mid$ $-$	Negation
$binop$ $::=$		
	$\mid$ $+$	Addition
	$\mid$ $*$	Multiplication
	$\mid$ $-$	Subtraction
	$\mid$ $<$	Less than
	$\mid$ $<=$	Less than equal
	$\mid$ $>$	Greater than
	$\mid$ $>=$	Greater than equal
	$\mid$ $==$	Equal
	$\mid$ $!=$	Not equal
	$\mid$ $\&\&$	And
	$\mid$ $\vert\vert$	Or

Appendix B

Side-Effect System

Typing Judgement

The side-effect system typing judgement takes the form:

$$\Gamma^{\mathrm{se}} \vdash e : \tau^{\mathrm{se}}, f$$

where

τ^{se}: base types enriched with latent side effects

$f \in \mathscr{P}(\mathit{Side\text{-}Effect})$: side-effect set

τ^{se}, f: representation of immediate and latent side effects

Γ^{se}: mapping from variables to τ^{se}

Typing Rules

For brevity, we write τ^{se} as τ and Γ^{se} as Γ. Note that these rules also capture file-reference tracking if we consider file references with different identifiers and groups (e.g. $\mathit{file\_ref}(i_1, g_1)$ and $\mathit{file\_ref}(i_2, g_2)$) to be of the same type.

$$\frac{}{\Gamma \vdash n : \mathit{int}, \{\}}\ (\mathit{int}) \qquad \frac{}{\Gamma \vdash b : \mathit{bool}, \{\}}\ (\mathit{bool}) \qquad \frac{}{\Gamma \vdash s : \mathit{string}, \{\}}\ (\mathit{string})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1 + e_2 : \mathit{int}, f_1 \cup f_2}\ (\mathit{add}) \qquad \frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1 * e_2 : \mathit{int}, f_1 \cup f_2}\ (\mathit{mult})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1 - e_2 : \mathit{int}, f_1 \cup f_2}\ (\mathit{sub}) \qquad \frac{\Gamma \vdash e_1 : \mathit{string}, f_1 \qquad \Gamma \vdash e_2 : \mathit{string}, f_2}{\Gamma \vdash e_1 + e_2 : \mathit{string}, f_1 \cup f_2}\ (\mathit{concat})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1 < e_2 : \mathit{bool}, f_1 \cup f_2}\ (<) \qquad \frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1 <= e_2 : \mathit{bool}, f_1 \cup f_2}\ (\leq)$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1 > e_2 : \mathit{bool}, f_1 \cup f_2}\ (>) \qquad \frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1 >= e_2 : \mathit{bool}, f_1 \cup f_2}\ (\geq)$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1 == e_2 : \mathit{bool}, f_1 \cup f_2}\ (=) \qquad \frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2}{\Gamma \vdash e_1\ != e_2 : \mathit{bool}, f_1 \cup f_2}\ (\neq)$$

$$\frac{\Gamma \vdash e_1 : \mathit{bool}, f_1 \qquad \Gamma \vdash e_2 : \mathit{bool}, f_2}{\Gamma \vdash e_1\ \&\&\ e_2 : \mathit{bool}, f_1 \cup f_2}\ (\wedge) \qquad \frac{\Gamma \vdash e_1 : \mathit{bool}, f_1 \qquad \Gamma \vdash e_2 : \mathit{bool}, f_2}{\Gamma \vdash e_1\ ||\ e_2 : \mathit{bool}, f_1 \cup f_2}\ (\vee)$$

$$\frac{\Gamma \vdash e : \mathit{bool}, f}{\Gamma \vdash !e : \mathit{bool}, f}\;(\neg) \qquad\qquad \frac{\Gamma \vdash e : \tau, f}{\Gamma \vdash (e) : \tau, f}\;(\mathit{paren})$$

$$\frac{}{\Gamma[x : \tau] \vdash x : \tau, \{\mathrm{R}, \mathrm{var}(x)\}}\;(\mathit{var\text{-}read}) \qquad \frac{\Gamma \vdash e_1 : \tau_1, f_1 \qquad (x : \tau_1), \Gamma \vdash e_2 : \tau_2, f_2}{\Gamma \vdash x := e_1;\ e_2 : \tau_2, (f_1 \cup f_2)}\;(\mathit{var\text{-}declare})$$

$$\frac{\Gamma \vdash e_1 : \tau_1, f_1 \qquad (x : \tau_1), \Gamma \vdash e_2 : \tau_2, f_2}{\Gamma[x : \tau_1] \vdash x = e_1;\ e_2 : \tau_2, (f_1 \cup f_2 \cup \{(\mathrm{W}, \mathrm{var}(x))\})}\;(\mathit{var\text{-}assign})$$

$$\frac{\Gamma \vdash e_1 : \mathit{unit}, f_1 \qquad \Gamma \vdash e_2 : \tau, f_2}{\Gamma \vdash e_1; e_2 : \tau, f_1 \cup f_2}\;(\mathit{seq}) \qquad \frac{\Gamma \vdash e : \mathit{unit}, f}{\Gamma \vdash \{e\}_{\{x,y,\ldots,z\}} : \mathit{unit}, f \setminus \{x, y, \ldots, z\}}\;(\mathit{block}^1)$$

$$\frac{\Gamma \vdash e_1 : \mathit{bool}, f_1 \qquad \Gamma \vdash e_2 : \mathit{unit}, f_2}{\Gamma \vdash \text{if } e_1 \ \{\, e_2 \,\} : \mathit{unit}, (f_1 \cup f_2)}\;(\mathit{if})$$

$$\frac{\Gamma \vdash e_1 : \mathit{bool}, f_1 \qquad \Gamma \vdash e_2 : \mathit{unit}, f_2 \qquad \Gamma \vdash e_3 : \mathit{unit}, f_3}{\Gamma \vdash \text{if } e_1 \ \{\, e_2 \,\} \text{ else } \{\, e_3 \,\} : \mathit{unit}, (f_1 \cup f_2 \cup f_3)}\;(\mathit{if\text{-}else})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}, f_1 \qquad \Gamma \vdash e_2 : \mathit{int}, f_2 \qquad \Gamma \vdash e_3 : \mathit{unit}, f_3}{\Gamma \vdash \text{for } x := e_1 \ ;\ x < e_2 \ ;\ x{+}{+} \ \{\, e_3 \,\} : \mathit{unit}, (f_1 \cup f_2 \cup f_3)}\;(\mathit{for\text{-}loop})$$

$$\frac{\Gamma \vdash e_1 : \mathit{string}, f_1 \qquad (x : \mathit{string}), \Gamma \vdash e_2 : \mathit{unit}, f_2}{\Gamma \vdash \text{for } x := \text{range } e_1 \ \{\, e_2 \,\} : \mathit{unit}, (f_1 \cup f_2)}\;(\mathit{for\text{-}each})$$

$$\frac{\Gamma \vdash e : \mathit{string}, f}{\Gamma \vdash \mathrm{print}(e) : \mathit{unit}, f \cup \{(\mathrm{W}, \mathrm{console})\}}\;(\mathit{print}) \qquad \frac{\Gamma \vdash n : \mathit{int}, \{\}}{\Gamma \vdash \mathrm{input}(n) : \mathit{string}, \{(\mathrm{W}, \mathrm{console})\}}\;(\mathit{input\text{-}1})$$

$$\frac{\Gamma \vdash n_1 : \mathit{int}, \{\} \quad \Gamma \vdash n_2 : \mathit{int}, \{\} \quad n_1 \leq n_2}{\Gamma \vdash \mathrm{input}(n_1, n_2) : \mathit{string}, \{(\mathrm{W}, \mathrm{console})\}}\;(\mathit{input\text{-}2}) \qquad \frac{\Gamma \vdash e : \mathit{string}, f}{\Gamma \vdash \mathrm{open}(e) : \mathit{file\_ref}(i', g'), f}\;(\mathit{open\text{-}1}^2)$$

$$\frac{\Gamma \vdash e : \mathit{string}, f \qquad \Gamma \vdash n : \mathit{int}, \{\}}{\Gamma \vdash \mathrm{open}(e, n) : \mathit{file\_ref}(i', n), f}\;(\mathit{open\text{-}2}^2) \qquad \frac{\Gamma \vdash e : \mathit{file\_ref}(i, g), f \qquad \Gamma \vdash n : \mathit{int}, \{\}}{\Gamma \vdash \mathrm{read}(e, n) : \mathit{string}, f \cup \{(\mathrm{R}, \mathrm{file}(i, g))\}}\;(\mathit{file\text{-}read\text{-}1})$$

$$\frac{\Gamma \vdash e : \mathit{file\_ref}(i, g), f \qquad \Gamma \vdash n_1 : \mathit{int}, \{\} \qquad \Gamma \vdash n_2 : \mathit{int}, \{\} \qquad n_1 \leq n_2}{\Gamma \vdash \mathrm{read}(e, n_1, n_2) : \mathit{string}, f \cup \{(\mathrm{R}, \mathrm{file}(i, g))\}}\;(\mathit{file\text{-}read\text{-}2})$$

$$\frac{\Gamma \vdash e_1 : \mathit{file\_ref}(i, g), f_1 \qquad \Gamma \vdash e_2 : \mathit{string}, f_2}{\Gamma \vdash \mathrm{write}(e_1, e_2) : \mathit{unit}, (f_1 \cup f_2 \cup \{(\mathrm{W}, \mathrm{file}(i, g))\})}\;(\mathit{file\text{-}write})$$

$$\frac{\Gamma \vdash e_1 : \mathit{file\_ref}(i, g), f_1 \qquad \Gamma \vdash e_2 : \mathit{string}, f_2}{\Gamma \vdash \mathrm{append}(e_1, e_2) : \mathit{unit}, (f_1 \cup f_2 \cup \{(\mathrm{W}, \mathrm{file}(i, g))\})}\;(\mathit{file\text{-}append})$$

$$\frac{x_1 : \tau_1, \ \ldots\ , x_n : \tau_n, \Gamma \vdash e : \tau, f \qquad (g : \tau_1 * \cdots * \tau_n \xrightarrow{f} \tau), \Gamma \vdash e' : \tau', f'}{\Gamma \vdash (\mathrm{def}\ g(x_1 : \tau_1, \ \ldots\ , x_n : \tau_n)\ \tau\ \{\, e \,\};\ e') : \tau', f'}\;(\mathit{def\text{-}func})$$

$$\frac{\Gamma(g) = \tau_1 * \cdots * \tau_n \xrightarrow{f} \tau \qquad \Gamma \vdash e_1 : \tau_1, f_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n, f_n}{\Gamma \vdash g(e_1, \ldots, e_n) : \tau, (f \cup f_1 \cup \cdots \cup f_n)}\;(\mathit{apply\text{-}func})$$

[1] All scopes remove local-variable side effects. However, we only apply this rule to the (*blocks*) for brevity.

[2] Where i' represents a fresh identifier and g' represents a fresh group.

Appendix C

Type-Cost System

Cost Definitions

In this section, we provide a full description of the *dependent cost calculus*: the representation of costs in Kautuka which depend on unknown input sizes. The theoretical definitions mirror my code implementation.

The set Var contains all program variables, specifically referring to *function inputs* in this context. When evaluated (dynamically), the value of a variable is its *size*. Exponentiated variables are represented with the set $\mathit{Exp\text{-}Var} = \mathit{Var} \times \mathbb{N}$, noting that exponents are constrained to *integers*[1]. This set contains elements of the form (x, n), representing exponentiated variables x^n. *Variable terms* are the product of an arbitrary number of integers and variables, and are written in the form $i \cdot x^m y^n \cdots z^p$ (where $i, m, n, \ldots, p \in \mathbb{N}$ and $x, y, \ldots, z \in \mathit{Var}$). This set is defined as $\mathit{Var\text{-}Term} = \mathbb{N} \times \mathscr{P}(\mathit{Exp\text{-}Var})$. These terms are summed to produce *polynomial terms* ($\mathscr{P}(\mathit{Var\text{-}Term})$) of the form $i \cdot w^m \cdots x^n + \cdots + j \cdot y^p \cdots z^q$. Note that this is simply a set-theoretical definition of *polynomial expressions*. In our language, polynomial terms are sufficient to describe all costs. A cost bound can be constructed with two polynomial terms ($\mathit{Poly\text{-}Term} \times \mathit{Poly\text{-}Term}$), representing the upper and lower bounds: $\langle \mathit{lower\_cost}, \mathit{upper\_cost} \rangle$. So a cost bound $\langle l, u \rangle$ contains the two *polynomial terms* l and u.

$$
\begin{aligned}
\mathit{Var} &= \{x, y, z, \ldots\} && x \\
\mathit{Exp\text{-}Var} &= \mathit{Var} \times \mathbb{N} && x^n \\
\mathit{Var\text{-}Term} &= \mathbb{N} \times \mathscr{P}(\mathit{Exp\text{-}Var}) && i \cdot x^m y^n \cdots z^p \\
\mathit{Poly\text{-}Term} &= \mathscr{P}(\mathit{Var\text{-}Term}) && i \cdot w^m \cdots x^n + \cdots + j \cdot y^p \cdots z^q \\
\mathit{Cost\text{-}Bound} &= \mathit{Poly\text{-}Term} \times \mathit{Poly\text{-}Term} && \langle \mathit{lower\_cost}, \mathit{upper\_cost} \rangle
\end{aligned}
$$

Cost Operations

Given two polynomial terms p_1 and p_2, we can perform the following operations: $(p_1 + p_2)$, $(p_1 - p_2)$, $(p_1 \cdot p_2)$, $\max(p_1, p_2)$, $\min(p_1, p_2)$.

Two *variable terms* are considered to be *matching* if they contain the same variables with the same exponents (but potentially differing integer coefficients). The operations $(p_1 + p_2), (p_1 - p_2), (p_1 \cdot p_2)$ are defined as expected (treating the polynomial terms as if they were polynomial expressions). For example, $(8x + 3xy^2) + (3x + 2x^2) = 11x + 3xy^2 + 2x^2$ and $(2x + 1) \cdot (3y) = 6xy + 3y$. This is implemented as follows:

When adding $p_1 + p_2$, we join together variable terms containing the same variables (for example $3xy + 5xy$ becomes $8xy$). This is implemented by scanning all variable terms in p_1 and p_2 and

[1] The language does not contain exponentiation operators, and we assume that loops do not produce exponential behaviour (section 2.5.1).

checking if they are matching. If they are, then we decompose the variable terms into their constituent parts: the variables $w^m \cdots x^n$ (shared between the terms) and the two coefficients of each term i_1 and i_2. We add $(i_1 + i_2) \cdot w^m \cdots x^n$ to the total (initially 0). Once we have identified all matching pairs, we then add all remaining variable terms in p_1 and p_2 to the result. Algorithm 2 presents an algorithm to implement this.

Algorithm 2: Addition of Polynomial Terms

```
Data: $p_1, p_2$
Result: $p$
$p \leftarrow 0$;
for $(i \cdot w^m \cdots x^n) \in p_1$ do
    $found\_matching \leftarrow false$;
    for $(j \cdot y^p \cdots z^q) \in p_2$ do
        if $is\_matching((i \cdot w^m \cdots x^n), (j \cdot y^p \cdots z^q))$ then
            $p \leftarrow p \cup \{((i + j) \cdot w^m \cdots x^n)\}$;
            $p_2 \leftarrow p_2 \backslash \{(j \cdot y^p \cdots z^q)\}$;
            $found\_matching \leftarrow true$;
        end
    end
    if $\neg found\_matching$ then
        $p \leftarrow p \cup \{(i \cdot w^m \cdots x^n)\}$;
    end
end
$p \leftarrow p \cup p_2$;
```

Subtraction follows similarly, where we instead add $((i - j) \cdot w^m \cdots x^n)$ to the result.

The multiplication of two *variable terms* $v_1 \cdot v_2$ is defined as multiplying their coefficients and unioning their exponentiated variables. However, if any of the variables in v_1 and v_2 match, then their exponents are added together during this process. This allows us to define the multiplication of two polynomial terms $p_1 \cdot p_2$: for all $v_1 \in p_1$ and $v_2 \in p_2$, we add $v_1 \cdot v_2$ to an initially empty result.

The minimum and maximum of two polynomial terms are defined conservatively. We say that the maximum of two different unknown variables x and y is $x + y$. But if two variable terms contain the same variables (and exponents) then we simply take the maximum of the coefficients, $\max(3xy^2 + 2x, 2xy^2 + 4y) = 3xy^2 + 2x + 4y$.

However, finding the minimum of two abstract terms is not possible with our current calculus. Taking $x - y$ does not work as we cannot ensure that the result is non-negative. One approach to this problem would be to add the min operator into the calculus, however extending the calculus in this way is outside the scope of this project. Another solution would be to approximate the minimum with an average of the two inputs, however our calculus only supports integers (and does not support rounding operations). Hence, we opt for an even cruder approach of approximating the minimum of two variables. We apply the same approach as calculating the maximum, however we take the minimum of coefficients rather than the maximum, $\min(3xy^2 + 2x, 2xy^2 + 2x + 4y) = 2xy^2 + 2x + 4y$. While this is not an accurate approximation, it does ensure that the result is always positive and is more meaningful than just returning 0. Surprisingly, the inaccuracy of this bound does not have a large impact overall (as shown by the results of section 4.2). The purpose of these bounds are to approximate the *magnitude* of costs, rather than to produce accurate estimates. This approach succeeds in that goal, however it is possible to construct adversarial inputs where this approach fails to produce reasonable estimates. We discuss the alternative approaches in further detail in section 5.2.

From these operations, we can define the cost definitions presented in the section 3.5.1:

$$\begin{aligned}
c_1 + c_2 &\triangleq \langle l_1 + l_2, u_1 + u_2 \rangle \\
c_1 \cdot c_2 &\triangleq \langle l_1 \cdot l_2, u_1 \cdot u_2 \rangle \\
c_1 \cup c_2 &\triangleq \langle \min(l_1, l_2), \max(u_1, u_2) \rangle \\
c_1 - c_2 &\triangleq \langle l_1 - u_2, u_1 - l_2 \rangle \\
c_1 \dot{-} c_2 &\triangleq \langle l_1 - l_2, u_1 - u_2 \rangle
\end{aligned}$$

Section 3.5.1 claims that: setting the type cost of an iterator variable i to the bound representing its range encapsulates all behaviour of a loop, even for variable values within the bound. We justify this with a proof which shows that all defined cost operations satisfy the following property: if $c_1 \subseteq c_2$, that is all values in the bound c_1 are contained within the bound c_2, then $(c_1 \oplus c) \subseteq (c_2 \oplus c)$ for operators $\oplus \in \{+, \cdot, \cup, -, \dot{-}\}$. This shows us that if operators are applied to both a value v (represented with the bound $\langle v, v\rangle$) and a range c encapsulating that value, then the value can never exceed the bound — hence the bound encapsulates all behaviour of the value. Note that we treat v as the bound $\langle v, v\rangle$ as these operators are only defined on cost bounds.

Theorem 1. *$c_1 \subseteq c_2 \implies (c_1 \oplus c) \subseteq (c_2 \oplus c)$ for all operators $\oplus \in \{+, \cdot, \cup, -, \dot{-}\}$. Where $\langle l_1, u_1\rangle \subseteq \langle l_2, u_2\rangle \iff l_1 \geq l_2 \wedge u_1 \leq u_2$.*

Proof. Proof by Exhaustion:

Let us define $c_1 = \langle l_1, u_1\rangle, c_2 = \langle l_2, u_2\rangle, c = \langle l, u\rangle$.

Case (+):

$$\begin{aligned}
c_1 \subseteq c_2 \quad = \quad & \langle l_1, u_1\rangle \subseteq \langle l_2, u_2\rangle \\
\implies \quad & l_1 \geq l_2 \wedge u_1 \leq u_2 \\
\implies \quad & l_1 + l \geq l_2 + l \wedge u_1 + u \leq u_2 + u \\
\implies \quad & \langle l_1 + l, u_1 + u\rangle \subseteq \langle l_2 + l, u_2 + u\rangle \\
= \quad & c_1 + c \subseteq c_2 + c
\end{aligned}$$

Cases ($\dot{-}$) and ($\cdot$) follow similar to above.

Case ($\cup$):

We note that $l_1 \geq l_2 \implies \min(l_1, l) \geq \min(l_2, l)$.

If $l_1 \leq l$, then $l_2 \leq l_1 \leq l$, hence $\min(l_1, l) = l$, $\min(l_2, l) = l$ where $l \geq l$ holds.

If $l_1 \geq l$, then $\min(l_1, l) = l_1 \geq l_2 \implies \min(l_1, l) \geq \min(l_2, l)$.

A similar logic can be used to prove $u_1 \leq u_2 \implies \max(u_1, u) \leq \max(u_2, u)$.

$$\begin{aligned}
c_1 \subseteq c_2 \quad = \quad & \langle l_1, u_1\rangle \subseteq \langle l_2, u_2\rangle \\
\implies \quad & l_1 \geq l_2 \wedge u_1 \leq u_2 \\
\implies \quad & \min(l_1, l) \geq \min(l_2, l) \wedge \max(u_1, u) \leq \max(u_2, u) \\
\implies \quad & \langle \min(l_1, l), \max(u_1, u)\rangle \subseteq \langle \min(l_2, l), \max(u_2, u)\rangle \\
= \quad & c_1 \cup c \subseteq c_2 \cup c
\end{aligned}$$

Case $(-)$:

$$\begin{aligned}
c_1 \subseteq c_2 &= \langle l_1, u_1 \rangle \subseteq \langle l_2, u_2 \rangle \\
&\implies l_1 \geq l_2 \wedge u_1 \leq u_2 \\
&\implies l_1 - u \geq l_2 - u \wedge u_1 - l \leq u_2 - l \\
&\implies \langle l_1 - u, u_1 - l \rangle \subseteq \langle l_2 - u, u_2 - l \rangle \\
&= c_1 - c \subseteq c_2 - c
\end{aligned}$$

Typing Judgement

The type-cost system typing judgement takes the form:

$$\Gamma^{\text{cost}} \vdash e : \tau^{\text{cost}}$$

where

τ^{cost}: type costs (unsized or quantified types)

Γ^{cost}: mapping from variables to τ^{cost}

Typing Rules

The following section lists the typing rules for the type-cost system which have not previously been mentioned throughout the dissertation. For brevity, we write τ^{cost} as τ and Γ^{cost} as Γ.

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \qquad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1 < e_2 : \mathit{bool}}\,(<) \qquad \frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \qquad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1 <= e_2 : \mathit{bool}}\,(\leq)$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \qquad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1 > e_2 : \mathit{bool}}\,(>) \qquad \frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \qquad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1 >= e_2 : \mathit{bool}}\,(\geq)$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \qquad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1 == e_2 : \mathit{bool}}\,(=) \qquad \frac{\Gamma \vdash e_1 : \mathit{int}(c_1) \qquad \Gamma \vdash e_2 : \mathit{int}(c_2)}{\Gamma \vdash e_1\, != e_2 : \mathit{bool}}\,(\neq)$$

$$\frac{\Gamma \vdash e_1 : \mathit{bool} \qquad \Gamma \vdash e_2 : \mathit{bool}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \mathit{bool}}\,(\wedge) \qquad \frac{\Gamma \vdash e_1 : \mathit{bool} \qquad \Gamma \vdash e_2 : \mathit{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \mathit{bool}}\,(\vee)$$

$$\frac{\Gamma \vdash e : \mathit{bool}}{\Gamma \vdash !e : \mathit{bool}}\,(\neg) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e) : \tau}\,(\mathit{paren})$$

$$\frac{}{\Gamma[x : \tau] \vdash x : \tau}\,(\mathit{var\text{-}read}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad (x : \tau_1), \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash x := e_1;\, e_2 : \tau_2}\,(\mathit{var\text{-}declare})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad (x : \tau_1), \Gamma \vdash e_2 : \tau_2}{\Gamma[x : \tau_1] \vdash x = e_1;\, e_2 : \tau_2}\,(\mathit{var\text{-}assign})$$

$$\frac{\Gamma \vdash e_1 : \textit{unit} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1; e_2 : \tau}\ (\textit{seq}) \qquad\qquad \frac{\Gamma \vdash e : \textit{unit}}{\Gamma \vdash \{e\} : \textit{unit}}\ (\textit{block})$$

$$\frac{\Gamma \vdash e_1 : \textit{bool} \qquad \Gamma \vdash e_2 : \textit{unit}}{\Gamma \vdash \text{if } e_1 \; \{\, e_2 \,\} : \textit{unit}}\ (\textit{if}^2)$$

$$\frac{\Gamma \vdash e_1 : \textit{string}(c) \qquad (x : \textit{string}\langle 1, 1\rangle), \Gamma \vdash e_2 : \textit{unit}}{\Gamma \vdash \text{for } x := \text{range } e_1 \, \{\, e_2 \,\} : \textit{unit}}\ (\textit{for-each}^2)$$

$$\frac{\Gamma \vdash e : \textit{string}(c)}{\Gamma \vdash \text{print}(e) : \textit{unit}}\ (\textit{print})$$

$$\frac{\Gamma \vdash e : \textit{string}(c)}{\Gamma \vdash \text{open}(e) : \textit{file\_ref}(i', g')}\ (\textit{open-1}) \qquad\qquad \frac{\Gamma \vdash e : \textit{string}(c) \qquad \Gamma \vdash n : \textit{int}\langle n, n\rangle}{\Gamma \vdash \text{open}(e, n) : \textit{file\_ref}(i', n)}\ (\textit{open-2})$$

$$\frac{\Gamma \vdash e_1 : \textit{file\_ref}(i, g) \qquad \Gamma \vdash e_2 : \textit{string}(c)}{\Gamma \vdash \text{write}(e_1, e_2) : \textit{unit}}\ (\textit{file-write})$$

$$\frac{\Gamma \vdash e_1 : \textit{file\_ref}(i, g) \qquad \Gamma \vdash e_2 : \textit{string}(c)}{\Gamma \vdash \text{append}(e_1, e_2) : \textit{unit}}\ (\textit{file-append})$$

$$\frac{x_1 : \tau_1, \, \ldots \, , x_n : \tau_n, \Gamma \vdash e : \tau_{\text{unsized}} \qquad (g : (x_1 : \tau_1^{\text{base}}) * \cdots * (x_n : \tau_n^{\text{base}}) \to \tau_{\text{unsized}}), \Gamma \vdash e' : \tau'}{\Gamma \vdash (\text{def } g(x_1 : \tau_1, \, \ldots \, , x_n : \tau_n) \; \tau \; \{\, e \,\}; \; e') : \tau'}\ (\textit{def-func-1})$$

$$\frac{\Gamma \vdash g : (x_1 : \tau_1^{\text{base}}) * \cdots * (x_n : \tau_n^{\text{base}}) \to \tau_{\text{unsized}} \qquad \Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash g(e_1, \ldots, e_n) : \tau_{\text{unsized}}}\ (\textit{apply-func-1})$$

[2] Where modification are made to the resultant Γ, as described in section 3.5.1.

Appendix D

Runtime-Cost System

Static Profiling

Some operations (such as addition) take practically no time to execute, no matter the input size. We estimate their execution time to be $1ns$, which is negligible compared to the operations we are interested in (with execution times of order $1\mu s$). However, we do not give them an execution time of $0ns$, as if these instructions are repeated many times (for example in deeply nested loops) their effect may become noticeable. These entries are marked as NEGL in table D.1.

I measured runtimes for each instruction, repeating the experiments 1000 times and calculating the average. For those instructions containing sized inputs, I varied the input size from 1 to $200,000$ and calculated the results for each. By plotting the relationship between input size and runtime using Python's `matplotlib`, I was able to determine the relationship between these variables: which in all cases was either constant or linear. For all linear relationships, I used `sklearn` to train a linear regression model on the data to extract its model parameters. The results of this analysis can be seen in the table below:

Instruction	Runtime ns	Instruction	Runtime ns
ADD(c_1, c_2)	NEGL	VAR_READ()	NEGL
MULT(c_1, c_2)	NEGL	VAR_DECLARE()	NEGL
SUB(c_1, c_2)	NEGL	VAR_ASSIGN()	NEGL
CONCAT(c_1, c_2)	NEGL	IF()	NEGL
LT()	NEGL	FOR_LOOP()	34
LE()	NEGL	FOR_EACH(c)	$11.7656c + 4.4181$
GT()	NEGL	PRINT(c)	$0.2118c + 486.8636$
GE()	NEGL	INPUT(c_1, c_2)[1]	3×10^9
EQUIV()	NEGL	FILE_OPEN()	4285
NE()	NEGL	FILE_READ(c)	$0.7596c + 78047.7545$
AND()	NEGL	FILE_WRITE(c)	$2.8451c + 1410315.1545$
OR()	NEGL	FILE_APPEND(c)	$2.3044c + 1310741.9063$
NOT()	NEGL	FUNC_CALL()	5

Figure D.1: Instruction runtime estimates measured on my machine.

We unusually treat the initialisation of the *for-each* structure as being dependent on the number of loop iterations. *For-each* loops in Go produce iterator variables of type *rune*. However, Kautuka only supports *strings*, so each loop is required to cast the iterator from a *rune* to a *string*. This takes non-negligible time, proportional to the number of loops performed.

[1] Modelled as a constant time of $3s$. It is impossible to estimate how long this will be, $3s$ was an arbitrarily picked value of the right order of magnitude.

Our linear models above produce float results, however our cost calculus only supports integers. We substitute the integer input sizes into the models, and round the result to the nearest integer. This allows us to produce more accurate estimates than if the model was constrained to only integer parameters (as the gradient is often very small).

The extra cost of parallelisation came to: $257.1065n + 674.2374$ where n is the number of blocks to be parallelised.

Typing Judgement

The runtime-cost system typing judgement takes the form:

$$\Gamma^{\text{run}} \vdash e : \tau^{\text{run}}, r$$

where

τ^{run}: type costs with latent runtime costs

$r \in$ *Cost-Bound*: runtime cost bound

τ^{run}, r: representation of immediate and latent runtime costs

Γ^{run}: mapping from variables to τ^{run}

Typing Rules

The following section lists the typing rules for the runtime-cost system which have not previously been mentioned throughout the dissertation. For brevity, we write τ^{run} as τ and Γ^{run} as Γ.

Note that the runtime-cost function for structures (e.g. IF(), FOR_LOOP(), etc.) refers to the runtime cost for initialising the structure, not the runtime of the structure itself.

$$\frac{}{\Gamma \vdash n : \mathit{int}\langle n, n\rangle, 0}\ (\mathit{int}^2) \qquad \frac{}{\Gamma \vdash b : \mathit{bool}, 0}\ (\mathit{bool}^2) \qquad \frac{\text{len}(s) = n}{\Gamma \vdash s : \mathit{string}\langle n, n\rangle, 0}\ (\mathit{string}^2)$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1), r_1 \quad \Gamma \vdash e_2 : \mathit{int}(c_2), r_2}{\Gamma \vdash e_1 - e_2 : \mathit{int}(c_1 - c_2), r_1 + r_2 + \text{SUB}(c_1, c_2)}\ (\mathit{sub})$$

$$\frac{\Gamma \vdash e_1 : \mathit{string}(c_1), r_1 \quad \Gamma \vdash e_2 : \mathit{string}(c_2), r_2}{\Gamma \vdash e_1 + e_2 : \mathit{string}(c_1 + c_2), r_1 + r_2 + \text{CONCAT}(c_1, c_2)}\ (\mathit{concat})$$

$$\frac{\Gamma \vdash e_1 : \mathit{int}(c_1), r_1 \qquad \Gamma \vdash e_2 : \mathit{int}(c_2), r_2}{\Gamma \vdash e_1 < e_2 : \mathit{bool}, r_1 + r_2 + \text{LT}(c_1, c_2)}\ (<)$$

[2] We assume that initialising values (which are known at compile time), has no cost.

Rules $(\leq)$,$(>)$,$(\geq)$,$(=)$,$(\neq)$ follow similarly to $(<)$.

$$\frac{\Gamma \vdash e_1 : \mathit{bool}, r_1 \qquad \Gamma \vdash e_2 : \mathit{bool}, r_2}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 : \mathit{bool}, r_1 + r_2 + \mathrm{AND}()}\,(\wedge) \qquad\qquad \frac{\Gamma \vdash e_1 : \mathit{bool}, r_1 \qquad \Gamma \vdash e_2 : \mathit{bool}, r_2}{\Gamma \vdash e_1 \mathbin{||} e_2 : \mathit{bool}, r_1 + r_2 + \mathrm{OR}()}\,(\vee)$$

$$\frac{\Gamma \vdash e : \mathit{bool}, r}{\Gamma \vdash !e : \mathit{bool}, r + \mathrm{NOT}()}\,(\neg) \qquad \frac{\Gamma \vdash e : \tau, r}{\Gamma \vdash (e) : \tau, r}\,(\mathit{paren}) \qquad \frac{}{\Gamma[x : \tau] \vdash x : \tau, \mathrm{VAR\_READ}()}\,(\mathit{var\text{-}read})$$

$$\frac{\Gamma \vdash e_1 : \tau_1, r_1 \qquad (x : \tau_1), \Gamma \vdash e_2 : \tau_2, r_2}{\Gamma \vdash x := e_1;\ e_2 : \tau_2, r_1 + r_2 + \mathrm{VAR\_DECLARE}()}\,(\mathit{var\text{-}declare})$$

$$\frac{\Gamma \vdash e_1 : \tau_1, r_1 \qquad (x : \tau_1), \Gamma \vdash e_2 : \tau_2, r_2}{\Gamma[x : \tau_1] \vdash x = e_1;\ e_2 : \tau_2, r_1 + r_2 + \mathrm{VAR\_ASSIGN}()}\,(\mathit{var\text{-}assign})$$

$$\frac{\Gamma \vdash e : \mathit{unit}, r}{\Gamma \vdash \{e\} : \mathit{unit}, r}\,(\mathit{block}) \qquad\qquad \frac{\Gamma \vdash e_1 : \mathit{bool}, r_1 \qquad \Gamma \vdash e_2 : \mathit{unit}, \langle l, u\rangle}{\Gamma \vdash \text{if } e_1 \ \{\ e_2\ \} : \mathit{unit}, e_1 + \langle 0, u\rangle + \mathrm{IF}()}\,(\mathit{if})$$

$$\frac{\Gamma \vdash e_1 : \mathit{string}(c), r_1 \qquad (x : \mathit{string}\langle 1, 1\rangle), \Gamma \vdash e_2 : \mathit{unit}, r_2}{\Gamma \vdash \text{for } x := \text{range } e_1 \ \{\ e_2\ \} : \mathit{unit}, r_1 + c \cdot r_2 + \mathrm{FOR\_EACH}(c)}\,(\mathit{for\text{-}each})$$

$$\frac{\Gamma \vdash e : \mathit{string}(c), r}{\Gamma \vdash \mathrm{print}(e) : \mathit{unit}, r + \mathrm{PRINT}(c)}\,(\mathit{print}) \qquad\qquad \frac{\Gamma \vdash n : \mathit{int}\langle n, n\rangle, 0}{\Gamma \vdash \mathrm{input}(n) : \mathit{string}\langle 0, n\rangle, \mathrm{INPUT}(\langle 0, n\rangle)}\,(\mathit{input\text{-}1})$$

$$\frac{\Gamma \vdash n_1 : \mathit{int}\langle n_1, n_1\rangle, 0 \qquad \Gamma \vdash n_2 : \mathit{int}\langle n_2, n_2\rangle, 0 \qquad n_1 \leq n_2}{\Gamma \vdash \mathrm{input}(n_1, n_2) : \mathit{string}\langle n_1, n_2\rangle, \mathrm{INPUT}(\langle n_1, n_2\rangle)}\,(\mathit{input\text{-}2})$$

$$\frac{\Gamma \vdash e : \mathit{string}(c), r}{\Gamma \vdash \mathrm{open}(e) : \mathit{file\_ref}(i', g'), r + \mathrm{FILE\_OPEN}()}\,(\mathit{open\text{-}1})$$

$$\frac{\Gamma \vdash e : \mathit{string}(c), r \qquad \Gamma \vdash n : \mathit{int}\langle n, n\rangle, 0}{\Gamma \vdash \mathrm{open}(e, n) : \mathit{file\_ref}(i', n), r + \mathrm{FILE\_OPEN}()}\,(\mathit{open\text{-}2})$$

$$\frac{\Gamma \vdash f : \mathit{file\_ref}(i, g), r \qquad \Gamma \vdash n : \mathit{int}\langle n, n\rangle, 0}{\Gamma \vdash \mathrm{read}(f, n) : \mathit{string}\langle 0, n\rangle, r + \mathrm{FILE\_READ}(\langle 0, n\rangle)}\,(\mathit{file\text{-}read\text{-}1})$$

$$\frac{\Gamma \vdash f : \mathit{file\_ref}(i, g), r \qquad \Gamma \vdash n_1 : \mathit{int}\langle n_1, n_1\rangle, 0 \qquad \Gamma \vdash n_2 : \mathit{int}\langle n_2, n_2\rangle, 0 \qquad n_1 \leq n_2}{\Gamma \vdash \mathrm{read}(f, n_1, n_2) : \mathit{string}\langle n_1, n_2\rangle, r + \mathrm{FILE\_READ}(\langle n_1, n_2\rangle)}\,(\mathit{file\text{-}read\text{-}2})$$

$$\frac{\Gamma \vdash e_1 : \mathit{file\_ref}(i, g), r_1 \qquad \Gamma \vdash e_2 : \mathit{string}(c), r_2}{\Gamma \vdash \mathrm{write}(e_1, e_2) : \mathit{unit}, r_1 + r_2 + \mathrm{FILE\_WRITE}(c)}\,(\mathit{file\text{-}write})$$

$$\frac{\Gamma \vdash e_1 : \mathit{file\_ref}(i, g), r_1 \qquad \Gamma \vdash e_2 : \mathit{string}(c), r_2}{\Gamma \vdash \mathrm{append}(e_1, e_2) : \mathit{unit}, r_1 + r_2 + \mathrm{FILE\_APPEND}(c)}\,(\mathit{file\text{-}append})$$

Let us say that expression e below returns expressions: $\tau_{\text{sized}}(c_1), \ldots, \tau_{\text{sized}}(c_n)$.

And $\tau_{\text{sized}}(c) = \tau_{\text{sized}}(c_1) \cup \cdots \cup \tau_{\text{sized}}(c_n)$.

(*def-func-2*):

$$\frac{x_1 : \tau_1(c_1), \ldots, x_n : \tau_n(c_n), \Gamma \vdash e : \tau_{\text{sized}}(c), r \qquad (g : \tau_1^{\text{base}} * \cdots * \tau_n^{\text{base}} \xrightarrow[g_{runtime}(c_1,\ldots,c_n)=r]{g_{size}(c_1,\ldots,c_n)=c} \tau^{\text{base}}), \Gamma \vdash e' : \tau', r}{\Gamma \vdash (\text{def } g(x_1 : \tau_1^{\text{base}}, \ \ldots \ , x_n : \tau_n^{\text{base}}) \ \tau^{\text{base}} \ \{ \, e \, \}; \, e') : \tau', r}$$

(*apply-func-2*[3]):

$$\frac{\Gamma \vdash g : (\tau_1^{\text{base}} * \cdots * \tau_n^{\text{base}}) \xrightarrow[g_{runtime}(c_1,\ldots,c_n)=r]{g_{size}(c_1,\ldots,c_n)=c} \tau^{\text{base}} \qquad \Gamma \vdash e_1 : \tau_1(c_1) \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n(c_n)}{\Gamma \vdash g(e_1, \ldots, e_n) : \tau_{\text{sized}}(g_{size}(c_1, \ldots, c_n)), g_{runtime}(c_1, \ldots, c_n)}$$

[3] We assume that initialising functions has no cost.

Appendix E

Parallelisation

This appendix details a proof outline for why algorithm 1 (presented in section 3.6.1) produces an *optimal* solution to block clustering (under the given assumptions and constraints).

The *interference operator* $\not\#$ is defined as the exact inverse of the non-interference operator #:

$$f_1 \not\# f_2 \leftrightarrow (\exists \sigma_1 \in f_1, \sigma_2 \in f_2 \cdot \sigma_1 \not\# \sigma_2)$$

This allows us to prove the following lemma:

Lemma 1. *If we have two non-interfering side-effect sets $f_1, f_2 \in \mathscr{P}(\text{Side-Effect})$, where f_1 is non-empty, then after removing a side effect from f_1 these side-effect sets are still non-interfering.*

$$(\sigma \cup f_1) \# f_2 \implies f_1 \# f_2$$

Proof. Proof by contrapositive: $f_1 \not\# f_2 \implies (\sigma \cup f_1) \not\# f_2$

Assume: $f_1 \not\# f_2$

$$\begin{aligned} f_1 \not\# f_2 &\implies \exists \sigma_1 \in f_1, \exists \sigma_2 \in f_2 \cdot \sigma_1 \not\# \sigma_2 && \textit{(By definition of } \not\# \textit{)} \\ &\implies \exists \sigma_1 \in (\sigma \cup f_1), \exists \sigma_2 \in f_2 \cdot \sigma_1 \not\# \sigma_2 \\ &\implies (\sigma \cup f_1) \not\# f_2 && \textit{(By definition of } \not\# \textit{)} \end{aligned}$$

We can also prove the following lemma regarding blocks and parallelisation groups:

Lemma 2. *Let us say we have a group G containing block A. If we were to move block A into a group later than G, then there are no blocks sequentially later than A which can be moved into group G (or a group preceding G), which would otherwise not be possible.*

Proof. Proof by contradiction.

Let us assume that placing block A into a group later than G now allows us to place block B into group G' (where G' is G, or a group preceding G). Note that B would be moved into a group which is executed sequentially earlier than the group it is currently in. This is because A (and hence G') is sequentially executed before B's original group.

The original statement says that, before moving A, it is not possible to place B into group G'. The only reason this could be is if B's side effects were to conflict with A's side effects, or with C's side effects, where C is a block between B and group G' (or C is in G'). If the former is the case, then moving A sequentially later still prevents us from moving B past A (as we cannot re-order blocks with conflicting side effects). Hence, if A is moved to a group later than G, than B cannot be placed to a group G or earlier. If the latter is the case, then moving A does not affect the position of C, so it would still be impossible to re-order B past C. So again, B cannot be placed into a group G or earlier. Hence, we derive a contradiction.

Here we provide a proof sketch for the *optimality* of algorithm 1:

Theorem 1. *Algorithm 1 produces an optimal parallelisation list (there does not exist another valid parallelisation list with less parallelisable groups). Algorithm 1 states that a block should be placed into a group at the earliest opportunity (without re-arranging or parallelising blocks with conflicting side effects).*

Proof. Proof by contradiction.

Let us say that not placing block A into group G at the earliest opportunity produces a more optimal solution. That is, placing A into a group later than G reduces the number of parallelisable groups in the program. Lemma 2 tells us that if we place block A into a group later than G, then G (and all groups preceding G) cannot contain any more blocks than would be possible if G did contain A. Hence, to prove that this produces a more optimal solution, placing A into a group later than G must reduce the total number of parallelisation groups past G, without any other blocks moved forward.

However, we now prove that if a construction of groups $\mathrm{G}_1, \ldots, \mathrm{G}_n$ contains A, then removing A does not invalidate this group. And hence there cannot exist a construction of groups with fewer parallelisation groups past G that becomes invalidated by removing A.

Let us say block A is now stored in block G' (sequentially later than block G). For the construction of groups to no longer be valid if A is removed, then the side-effects in group G' must conflict if A is removed. However, (Lemma 1) proves that: if two side-effect sets are non-interfering and a side effect is removed from one of these side-effect sets then they are still non-interfering. Since the side effects in G' (with A) were non-interfering, then the removal of A's side-effects means that these side effects must still be non-interfering. Hence, there cannot be a construction of groups which becomes invalidated if A is removed, and so we derive a contradiction.

Appendix F

Proposal

1 Introduction

Since 2006, improvements made to single-core processor performance have begun to stagnate as we begin to approach theoretical limits in advancements to transistor technology. This has led to an uptake in parallel processing, the act of splitting computations across multiple processors running simultaneously, in order to increase computational throughput by utilising more than one processor at once. However, there have been some limiting factors in the progress of parallel processing - the most prominent of which being the difficulties humans face when attempting to reason about concurrent systems (we assume that concurrency refers specifically to parallelism concurrency in this proposal). Human brains seem to be designed for sequential calculations [1] which makes it very hard for us to design and understand systems containing multiple threads of independent, interlocking calculations which are running in parallel. In order to try and aid programmers, modern programming languages often include concurrency primitives to reduce the cognitive load needed to design these sorts of systems, but despite this, many programmers still find concurrency difficult and often opt to avoid it when it is not strictly necessary. This may cause problems in the future as programmers will either have to embrace concurrency, despite its current flaws, or potentially miss out on the yearly performance improvements in computation which have been observed in previous decades.

In this project, I propose a new sequential programming language and a compiler, written in OCaml, which compiles this sequential language to parallelised code at the function level. This means that users can gain the benefits of parallel programming while avoiding many complexities surrounding concurrency. I will design a simple sequential programming language with Go-like syntax, which is provisionally dubbed Kautuka after the woven Indian "thread"[1]. I will compile Kautuka to the Go programming language, automatically parallelising the code during this compilation process using the Go's concurrency primitives: channels and goroutines[2].

When compiling from sequential Kautuka to parallel Go code, we expect that the parallel code acts in the same way as the sequential code, regardless of any non-determinism introduced by threads. Hence any instructions which may be affected by running in parallel, namely effects, should be statically analysed before compilation to ensure that effects in different threads do not interfere. And secondly, we expect that the compiler makes an attempt to produce parallel Go code which is computationally faster than the equivalent sequential Go code. Hence we introduce a static analysis: effect tracking and cost analysis into the compiler to ensure these expectations are met. As part of this project, I will implement a novel approach to cost analysis by designing a type system based upon tracking lower and upper bounds of values. My approach to cost analysis and effect tracking is explored in further detail in the next section.

[1] `https://en.wikipedia.org/wiki/Kautuka`

[2] Goroutines are lightweight threads managed by the Go runtime designed to run functions in parallel

2 Structure of the Project

There are five main components to this project:

- Translation: Building a compiler from Kautuka to sequential Go
- Effect tracking: Implementing effect tracking into Kautuka
- Cost analysis: Implementing cost analysis into Kautuka
- Automatic parallelisation: Extending the Translation step to compile Kautuka to parallelised Go based upon the previous two steps
- Evaluation: refer to Evaluation section

Translation

This involves designing the syntax and semantics of my custom language: Kautuka. Once the syntax and semantics have been formalised, I can use OCaml's lexing and parsing libraries to compile Kautuka to an AST, along with tests to ensure this has been carried out correctly. Finally, I can develop a tool which converts this AST to Go code, and write tests to check that the produced Go code is semantically equivalent to the original Kautuka program.

Effect tracking

I will extend the compiler by introducing a basic effect system based upon [2]. This will allow me to track the effects which may potentially be introduced by calls to user-defined functions, in order to ensure that parallel threads do not contain conflicting effects. The core project will only focus on the effects of mutating non-local state, but this can be extended to reasoning about console IO and file system effects in my project extensions. However, this project will not focus on pointers and aliases, and so we will assume that pointers of the same type will conflict unless we are explicitly told otherwise by the user.

Cost analysis

A large part of my project will be implementing an original approach to cost analysis by designing a custom type system which allows us to better estimate the time it takes to run functions. The type system will track the minimum and maximum value of variables[3] (henceforth referred to as bounds), and uses inference to update these bounds throughout the program. By measuring the execution time of standard functions for different input size, we can estimate the minimum and maximum execution time of all the standard function calls in a given program. The novelty of this idea is that tracking the lower and upper bounds of variables does not seem to have been used to statically estimate the execution times of functions before. However, tracking the intervals of integer variables seems to be well researched in the field of scientific computing and so I intend to apply some of these ideas to the cost analysis in my project. When determining the runtime of all user-declared functions in the program, I intend to use [3] as a reference for dealing with conditions and loops. Hence, this analysis can

[3] Value refers to the numerical value for int types and size for string and array types

be used to estimate how much time would be saved or lost by parallelising sets of functions, taking into consideration any costs incurred by introducing threads. The main tradeoff for my proposed approach is that Kautuka's type system is quite verbose and require a lot of extra type annotations from the user surrounding estimated bounds of variables. In my extensions, we can explore cost analysis in relation to higher-order and first-class functions, also using [3] as a reference, which will add significant complexity to this analysis.

Automatic parallelisation

Effect tracking allows us to identify combinations of function sets which do not contain any interfering effects. We can then use the cost analysis to determine the best way to schedule functions into separate threads in order to gain the most benefit; this involves determining whether parallelisation will give us any benefit at all compared to compiling functions purely sequentially. If we decide to carry out parallelisation, we will compile the program to Go; making use of goroutines to parallelise functions and channels to pass results out of successfully terminated goroutines (similar to *futures*[4]).

3 Evaluation

Quantitative

Evaluate the execution time of Kautuka compared to the equivalent sequential Go code. The examples will be handcrafted and well-suited to potential parallelisation optimisation as that is the intended use case of my program.

Qualitative

Ensure that my compiled language is semantically equivalent to the sequential Go code it is emulating, using the same corpus of examples as above.

4 Starting Point

I have limited prior experience in implementing lexers and parsers, and no prior experience in implementing effect systems or cost analysis. However I have studied Semantics of Programming Languages and Compiler Construction which should aid with building the lexer and parser stages of my compiler. Furthermore I have read ahead on the Optimising Compilers course regarding effect systems.

5 Success Criteria

For the project to be deemed a success, the following must be successfully completed:

[4] `https://en.wikipedia.org/wiki/Futures_and_promises`

1. Design the syntax for Kautuka and then develop a compiler from Kautuka to Go. Kautuka should support basic control flow, iterators and (non-first-class and non-higher-order) functions. My compiler will not include type checking, this stage will be carried out when the Go code is compiled further using the standard Go compiler.
2. Implement effect tracking for Kautuka. In the core project, this is limited to tracking modifications of non-local state.
3. Implement cost analysis for Kautuka. This involves creating a type system to track the minimum and maximum values of variables, and then measuring the runtime of standard functions based upon inputs of different sizes. Hence we can calculate the estimated minimum and maximum runtime of functions.
4. Parallelise the code during the compilation process based upon this analysis. We want to compile functions into parallel threads if there are no interfering effects and we deem there to be sufficient cost benefit.
5. Compare the runtime of a Kautuka program compiled to parallelised Go with the equivalent sequential Go code. And also evaluate whether a given Kautuka program is semantically equivalent to the Go code it is compiled to.

6 Possible Extensions

Possible extensions include:

1. Add IO support to Kautuka, namely interactions with console IO and the file system.
2. Add support for first-class functions and higher-order functions to Kautuka.
3. Add support for data structures, such as structs and enums, to Kautuka.
4. Seamlessly integrate Kautuka programs with existing Go codebases. This involves developing a tool that allows us to run a project containing both Kautuka and Go files using a combination of both my compiler and the standard Go compiler.

7 Timetable and Milestones

Weeks 1 to 2 (17 Oct 22 - 30 Oct 22)

Proposal Submitted

Learn best software development practices in OCaml and research tools available for creating the frontend of a compiler (e.g. lexer and parser tools).

Carry out research into effect tracking and cost analysis. For effect tracking, re-read the Optimising Compilers lectures regarding effects systems [2] and carry out further reading. For cost analysis, read [3] and research other papers for more inspiration on how best to implement this into my compiler.

Familiarise myself with Go syntax. Begin designing the syntax for Kautuka.

Begin writing the Introduction draft in dissertation.

Weeks 3 to 4 (31 Oct 22 - 13 Nov 22)

Finish designing the syntax for Kautuka.

Implement the lexer and parser for the language and write tests to verify that the output abstract syntax trees are correct. Write a compiler to convert the output abstract syntax tree into Go code, which at this point will closely resemble the source code we started with.

Finish writing the Introduction draft in dissertation.

Milestone: Given a Kautuka program, generate the corresponding abstract syntax tree

Weeks 5 to 6 (14 Nov 22 - 27 Nov 22)

Design and formalise effect tracking for my language. Then implement this into the compiler to generate lists of effects for all functions in the program. Add more tests to verify that this has been done correctly.

Begin writing the Implementation draft in dissertation (specifically adding my formalised Kautuka syntax and effect tracking inference rules).

Milestone: Given a Kautuka program, determine the list of effects for all of the functions

Weeks 7 to 8 (28 Nov 22 - 11 Dec 22)

End of Michaelmas Term - start of Christmas holidays.

Design and formalise my proposed bound-based type system. Then implement this into the compiler. Collect data surrounding the execution time for the standard functions and the overheads involved with creating threads.

Continue writing the Implementation draft in dissertation.

Milestone: Extend Kautuka with my bound-based type system, represent these bounds in the generated abstract syntax tree

Weeks 9 to 11 (12 Dec 22 - 1 Jan 22)

Finish collecting data surrounding execution times and create an algorithm to predict how long functions will take on inputs of different sizes. Using my type system and the collected data, now extend the compiler to estimate the runtime of the user's functions. Finally we can compile functions into different threads if there is sufficient cost benefit.

Start my evaluation.

Finish writing the Implementation draft in dissertation.

Take time off for Christmas

Milestone: Estimate the runtime of all user defined Kautuka functions. Compile Kautuka to parallelised Go based upon the analysis

Weeks 12 to 13 (2 Jan 22 - 15 Jan 22)

Add better type system inference and create further tests to ensure that the program is compiled correctly.

Evaluate the runtime of Kautuka compared to the semantically equivalent, sequential Go code.

Begin writing the Evaluation draft in dissertation.

Milestone: Finish core project implementation and pass all success criteria

Weeks 14 to 15 (16 Jan 22 - 29 Jan 22)

End of Christmas holidays - start of Lent Term.

Slack time to finish core project implementation.

Make a start on progress report.

Weeks 16 to 17 (30 Jan 22 - 12 Feb 22)

Finish progress report for the deadline Fri 3 Feb 2023. And create progress report presentation for the deadline Wed 9 Feb 2023.

Finish writing the Evaluation draft in dissertation.

Milestone: Submit progress report

Weeks 18 to 19 (13 Feb 22 - 26 Feb 22)

Make a start on extensions if time permits. Mainly focus on extensions 1 and 4.

Start Conclusion draft in dissertation.

Weeks 20 to 21 (27 Feb 22 - 12 Mar 22)

Continue with extensions if time permits. Now focus on the remaining extensions.

Finish Conclusion draft in dissertation.

Milestone: Finish dissertation draft

Weeks 22 to 23 (13 Mar 22 - 26 Mar 22)

Slack time to finish the dissertation draft.

Weeks 24 to 25 (27 Mar 22 - 9 Apr 22)

Submit draft of the dissertation to my supervisor and DoS for review by the deadline Fri 7 Apr 2023. Begin exam revision and review code repository in the meantime (ensure that my code is clear, readable and well structured).

Make improvements to the dissertation based upon this feedback.

Weeks 26 to 27 (10 Apr 22 - 23 Apr 22)

Finish improvements to the dissertation and submit to my supervisor a second time for final review.

Make last adjustments and focus on making the dissertation presentable and easy to read.

Weeks 28 - 29 (24 Apr 22 - 7 May 22)

Slack time for any extra additional evaluation and tests that may be discovered during final reviews of my dissertation, and start revision for exams.

Milestone (28 April 22): Submit Dissertation (2 weeks in advance of final deadline)

8 Resource Declaration

I will be using my personal laptop (Dell XPS 15 7590 - 2.6GHz i7, 16GB RAM) as my primary machine for software development. As a backup, I will use my secondary personal laptop and resources provided by SRCF (student run computing facility). I will continuously backup my code and dissertation with Git version control and carry out periodic backups to Google Drive.

References

[1] Mariano Sigman and Stanislas Dehaene. “Brain Mechanisms of Serial and Parallel Processing during Dual-Task Performance”. In: *Journal of Neuroscience* 28.30 (2008), pp. 7585–7598. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.0948-08.2008. eprint: https://www.jneurosci.org/content/28/30/7585.full.pdf. URL: https://www.jneurosci.org/content/28/30/7585.

[2] Timothy M. Jones. *Optimising Compilers Lecture 13 Effect Systems*. 2022. URL: https://www.cl.cam.ac.uk/teaching/2122/OptComp/slides/lecture13.pdf (visited on 10/07/2022).

[3] Ben Wegbreit. “Mechanical Program Analysis”. In: *Commun. ACM* 18.9 (Sept. 1975), pp. 528–539. ISSN: 0001-0782. DOI: 10.1145/361002.361016. URL: https://doi.org/10.1145/361002.361016.