Compiler Optimisations for High-Level Synthesis

Optimisations from SCI Inference

Yu-Yang Lin
1228863
MSci in Computer Science

Final Year Project Dissertation
Supervised by Dr. Dan R. Ghica

School of Computer Science
University of Birmingham
United Kingdom
4th April, 2016

Abstract

This report will describe the work involved in designing and implementing a compiler optimisation. The optimisation will require the SCI typing framework and will be applied to a High-Level Synthesis compiler for hardware architectures such as field-programmable gate array (FPGA). The optimisation presented will be a non-standard optimisation designed specifically to tackle call-by-name evaluation and concurrency in hardware architectures. The specific problem tackled is the inefficiency of potential argument reevaluation with call-by-name. To fix reevaluation, I propose and implement an optimisation that caches arguments while employing parallelisation in order to take advantage of FPGA architecture.

All the software written for this project can be found at:
https://codex.cs.bham.ac.uk/svn/projects/2015/yxl265

To my family and friends

Acknowledgements

I want to thank Dan for supervising my project and for providing feedback and suggestions. I would also like to thank Alex and Vasil for the discussion during meetings—as well as some other technical help—and thank Bertie for helping me understand how to use his SCI inference optimisation framework.

Contents

Chapter 1

Introduction

In this report I describe the work involved in designing and implementing an SCI-based optimisation for High-Level Synthesis (HLS). The goal of the optimisation described in this report is to solve two problems specific to the hardware compiler used in this project: first is the motivation behind fine-grained parallelisation in hardware; and second is the problem with reevaluation of arguments in call-by-name strategies.

The project report will be structured as follows: I will first provide the required background knowledge to understand the optimisation; this is followed by the rationale and description of the algorithm and how it was designed; then an outline of the implementation is given; and finally an evaluation of the optimisation is done. Evaluation will be done using sample programs, and the output of optimising said programs will also be provided. The following section will thus start with the background knowledge.

1.1 Reynolds' Syntactic Control of Interference (SCI and SCI2)

In 1978, John Reynolds (POPL, 1978)[3] proposed a typing framework that could control interference through syntactic restrictions. The rationale is that any imperative language that is powerful enough (capable of assignments and procedures) would inevitably produce interference. Though vaguely described as "interfering side effects", the actual implication of interference involves more subtleties. In his paper, Reynolds describes interference between two *phrases* (commands or procedures) to exist if executing either phrase can change the value of the other.

The key aspect of the paper was the proposal of three design principles for the linguistic framework that would allow interference to be syntactically detectable. These three principles (or constraints) are the following:

- If no free identifier in phrase p interferes with any free identifier in phrase q, then p does not interfere with q.

  This means, all interference "channels" are known and named by the identifiers, reducing the problem of interference between two phrases to identifying whether free variables in the phrases interfere.

- Distinct identifiers do not interfere.

  This also implies that two disjoint sets of identifiers are known to not interfere.

- Side-effect free (*passive*) phrases do not interfere with one another.

This means that sets of passive identifiers do not have to be disjoint to be known to not interfere.

However, Reynolds then shows that the syntactic constraints defined above have the unfortunate property that β-reduction of legal phrases may produce illegal phrases. In a following proposal (ICALP, 1989)[4], Reynolds presents a solution for this problem using intersection (conjunctive) types to define constraints.

To describe this typing framework, Reynolds provided a sample syntax for phrases and the variables he used with their meaning.

1.1.1 Sample untyped abstract syntax for phrases

$\langle \text{phrase} \rangle ::= \langle \text{identifier} \rangle$	identifiers
$\mid \lambda \langle \text{identifier} \rangle : \langle \text{finite set of types} \rangle . \langle \text{phrase} \rangle$	abstraction
$\mid \langle \text{phrase} \rangle \ \langle \text{phrase} \rangle$	application
$\mid \langle \langle \text{identifier} \rangle \equiv \langle \text{phrase} \rangle, \ldots, \langle \text{identifier} \rangle \equiv \langle \text{phrase} \rangle \rangle$	tupling
$\mid \langle \text{phrase} \rangle . \langle \text{identifier} \rangle$	selection
$\mid \textbf{if} \langle \text{phrase} \rangle \textbf{then} \langle \text{phrase} \rangle \textbf{else} \langle \text{phrase} \rangle$	conditionals
$\mid 0 \mid 0.5 \mid \langle \text{phrase} \rangle + \langle \text{phrase} \rangle$	expressions
$\mid \langle \text{phrase} \rangle := \langle \text{phrase} \rangle \mid \langle \text{phrase} \rangle ; \langle \text{phrase} \rangle \mid \textbf{while} \langle \text{phrase} \rangle \textbf{do} \langle \text{phrase} \rangle$	commands
$\mid \langle \text{phrase} \rangle \mid\mid \langle \text{phrase} \rangle$	concurrency

In short, phrases can be commands—primitive constructs like assignment, sequencing, loops—and procedures, which contain expressions and other phrases. A especially important one here is concurrency. The operator $(||)$ states that two phrases run in parallel. This will be especially useful in hardware compilation alter on since in hardware, we aim towards parallelism.

1.1.2 Types

Reynolds describes the following types:

- *data type*—such as "integer" and "boolean", denoting a kind of variable or expression;
- *phrase type*—such as "integer expression" and "procedure accepting an integer expression", which denotes a set of meanings for a kind of phrase.

The metavariables used in his paper are the following:

- δ : data types
- ϕ : passive phrase types
- α : active phrase types
- θ : arbitrary phrase types
- $\hat{\phi}$: finite set of passive phrase types
- $\hat{\theta}$: finite set of arbitrary phrase types
- ι : identifiers

Additionally, we have:

- $\hat{\theta} \to \theta$: arbitrary procedure
- $\hat{\theta} \xrightarrow{P} \theta$: passive procedure
- δ **exp** : an expression phrase of evaluated type δ
- δ **acc** : an acceptor phrase which can accept a value of type δ when executed
- **comm** : a primitive phrase type for commands

1.1.3 Inference rules

With these definitions, inference rules were given for: identifiers, subtypes, abstraction, application, tupling, field selection, conditionals, arithmetic expressions, commands, and concurrency. However, the need for subtyping made the rules substantially more complicated than the original type system. The following section will thus describe an alternative approach.

1.2 SCI Revisited [2]

Inference rules for the SCI2 type system depend on subtyping and intersection types. This is a problem as the rules were already fairly complicated; further adding to the complications of intersection types. O'Hearn *et al.* tackle this with SCIR, in which they propose a simpler set of rules which do not require subtyping.

1.2.1 The SCIR type system

Rules defined have two zones, a passive and an active zone. This is denoted by $(\Pi \mid \Gamma)$.

Identity Rule

$$\frac{}{\mid \iota : \theta \vdash \iota : \theta} \quad \text{Axiom}$$

Structural Rules

$$\frac{\Pi \mid \Gamma, \iota : \theta \vdash P : \phi}{\Pi, \iota : \theta \mid \Gamma \vdash P : \phi} \quad \text{Passification}$$

$$\frac{\Pi, \iota : \theta \mid \Gamma \vdash P : \theta'}{\Pi \mid \Gamma, \iota : \theta \vdash P : \theta'} \quad \text{Activation}$$

$$\frac{\Pi \mid \Gamma \vdash P : \theta}{\Pi, \Pi' \mid \Gamma, \Gamma' \vdash P : \theta} \quad \text{Weakening}$$

$$\frac{\Pi \mid \Gamma \vdash P : \theta}{\widetilde{\Pi} \mid \widetilde{\Gamma} \vdash P : \theta} \quad \text{Exchange}$$

$$\frac{\Pi, \iota : \theta, \iota' : \theta \mid \Gamma \vdash P : \theta'}{\Pi, \iota : \theta \mid \Gamma \vdash [P](\iota' \mapsto \iota) : \theta'} \quad \text{Contraction}$$

Type Constructor Rules

$$\frac{\Pi \mid \Gamma \vdash P : \theta_0 \qquad \Pi \mid \Gamma \vdash P : \theta_1}{\Pi \mid \Gamma \vdash \langle P, Q \rangle : \theta_0 \times \theta_1} \quad \times I$$

$$\frac{\Pi \mid \Gamma \vdash P : \theta_0 \times \theta_1}{\Pi \mid \Gamma \vdash \pi_i P : \theta_i} \quad \times E_i (i = 0, 1)$$

$$\frac{\Pi_0 \mid \Gamma_0 \vdash P : \theta_0 \qquad \Pi_1 \mid \Gamma_1 \vdash P : \theta_1}{\Pi_0, \Pi_1 \mid \Gamma_0, \Gamma_1 \vdash \langle P, Q \rangle : \theta_0 \otimes \theta_1} \quad \otimes I$$

$$\frac{\Pi \mid \Gamma \vdash P : \theta_0 \otimes \theta_1 \qquad \Pi' \mid \Gamma', \iota_0 : \theta_0, \iota_1 : \theta_1 \vdash Q : \theta}{\Pi, \Pi' \mid \Gamma, \Gamma' \vdash \textbf{let } \iota_0 \otimes \iota_1 \textbf{ be } P \textbf{ in } Q : \theta} \quad \otimes E$$

$$\frac{\Pi \mid \Gamma, \iota : \theta' \vdash P : \theta}{\Pi \mid \Gamma \vdash \lambda \iota : \theta' . P : \theta' \to \theta} \quad \to I$$

$$\frac{\Pi_0 \mid \Gamma_0 \vdash P : \theta' \to \theta \qquad \Pi_1 \mid \Gamma_1 \vdash Q : \theta'}{\Pi_0, \Pi_1 \mid \Gamma_0, \Gamma_1 \vdash P(Q) : \theta} \quad \to E$$

$$\frac{\Pi \mid \vdash Q : \theta' \to \theta}{\Pi \mid \vdash \textbf{promote } Q : \theta' \xrightarrow{P} \theta} \quad \xrightarrow{P} I$$

$$\frac{\Pi \mid \Gamma \vdash Q : \theta' \xrightarrow{P} \theta}{\Pi \mid \vdash \textbf{derelict } Q : \theta' \to \theta} \quad \xrightarrow{P} E$$

Passification and *Activation* are actions applicable to identifiers which move them into or out of the passive zone respectively. *Weakening* introduces unused assumptions, *exchange* implies order does not matter since $\tilde{\Pi}$ and $\tilde{\Gamma}$ are permutations of Π and Γ—both can be applied in both active and passive zones. *Contraction* tells us when we can reduce the set of distinct identifiers occurring in a phrase. Hence it is one of the driving rules making the type system appealing—interference occurs in "channels" named by identifiers, meaning the problem of detecting interference between two phrases is simplified to ensuring active identifiers do not occur freely in both phrases.

Dereliction refers to going from well-behaved to bad-behaved, *promotion* goes in reverse. Going from well-behaved (passive) to bad-behaved (active) has no constraints, *promotion*, conversely, can only be applied to passive zones.

1.2.2 SCI2 vs. SCIR

The advantage of SCIR is that it does not require subtypes and other intersection types. In exchange it has an apparent problem with completeness—SCIR cannot type check some programs that SCI2 can.

For example,

$$\lambda c_1 : \textbf{comm} . \lambda c_2 : \textbf{comm} . \lambda c_3 : \textbf{comm} . \pi_0 \langle c_1, c_3 \rangle \,||\, \pi_0 \langle c_2, c_3 \rangle \\ : \textbf{comm} \to \textbf{comm} \to \textbf{comm} \to \textbf{comm}$$

cannot be type checked because c_3—though unused—is an active identifier that occurs on both sides of the parallel composition. Since an active identifier is present on both sides, the phrases

cannot run in parallel. This is not a problem in SCI2 because unused parts of a product will become passive.

In the paper, an argument is given suggesting that this example points at the inability of SCIR to do subtyping rather than at its incompleteness. The reasoning is that—though unformulated—SCIR with intersection types is a possibility.

1.3 SCI in High-Level Synthesis

Interference is not just a source of programming errors, but directly impacts type structure and parallelism of a program. The proposal was that SCI would hopefully ease reasoning about interference in imperative programs, as well as providing the framework that will enforce syntactic restrictions using information about interference of phrases. It is the later property which is of particular interest with in this project. Understanding interference results in understanding phrase parallelism, which is useful for optimisations. By controlling interference syntactically, we are provided a framework that tells us when two phrases might interfere, meaning, when optimisations that depend on parallelism are applicable.

Hence the optimisations implemented will be SCI specific—these rely heavily on parallelisation and caching of variables so understanding phrase parallelism and interference is essential. The effect of parallelisation-based optimisations in hardware is made clearer when contrasting concurrency in a CPU with hardware architectures such as with field-programmable gate arrays (FPGA). In CPUs, parallelisation is an expensive and difficult problem. The common CPU solution is to use threads in a process.

These, however, are expensive:

- there is overhead in creating a new thread for each phrase executed in parallel;
- there is overhead in managing threads in a CPU, there are many expensive procedures taking part to achieve this; from pipelining to context-switching, to instruction-level parallelisation in superscalar processors, achieving parallelisation in an optimal way can be tricky. Context-switching on its own is already a big factor–it is pure overhead as the CPU cannot perform any useful work while switching.

In contrast, hardware does parallelisation very cheaply. Hardware compilation allows us to simply run new physical signals concurrently for every phrase composed in parallel. Meaning parallelisation has no overhead other than the physical space required to run a "wire". Hence, SCI allows for optimisations especially useful in High-Level Synthesis. In fact, the optimisations implemented can even be detrimental with CPU-based systems due to overhead of parallelisation. As such, the optimisation described here will not be standard; rather, it is very specific to the compiler I will be using. Out of context, it is in fact a *pesimisation* and should not be used.

1.4 Call-by-name Evaluation

Call-by-value is not suitable with a hardware compilation language. This is because with hardware, things like garbage collection are a problem. Thus, a static memory model is preferred and call-by-value cannot ensure this. Call-by-need, contrastingly, seems to be compatible with hardware-oriented languages. Efficient implementation on hardware is still an area of ongoing research, however. Languages like Haskell cache arguments to optimise evaluation, the complication is that the strategy is incompatible with side-effects. Haskell gets around this by the

use of monads to globalise effects. This is not an ideal solution as we are interested in localised effects—which Reynold's SCI provides.

Alternatively, call-by-name provides a static memory model that is compatible with localised effects. Hence, it is more suitable than the other evaluation strategies mentioned and is, in fact, the strategy used by the language I will be working with. The problem with this evaluation strategy is that it can be potentially inefficient due to reevaluation. This is fine if arguments are not used in functions—in fact better than call-by-value since no arguments are ever evaluated then—however, this property only applies to a small subset of useful programs.

1.5 Geometry of Synthesis [1]

Geometry of Synthesis (GOS) approaches higher-level synthesis in a way such that functions are fully implemented as would be in ordinary—non hardware-oriented—programming languages. This means function calls are handled fully rather than through inlining—as is typically done in HLS, allowing us to explore algorithms which benefit from dynamic function calls and recursion.

I will be working with the GOS compiler in this project. The compiler itself is written in OCaml and the programming language for it is called *Verity*. This is an Algol-like language based on Reynold's idealised concurrent Algol. Hence Verity:

- uses call-by-name Lambda calculus for its functional fragment—which is observationally equivalent to call-by-need;
- has higher-order recursion;
- uses a localised effects for its imperative fragment—which will be related to SCI.

Verity does not have pointers, allocation, or garbage collection; given memory can be on chip, there are no memory management primitives provided. In hardware there isn't really a global memory to manipulate anyway. Additionally, the language is almost completely transparent to the programmer with respect to hardware specific constructs. The only hardware-specific requirement is the definition of bit-widths for integer expressions.

1.6 The SCI Implementation and Optimisation Framework

The optimisations I will be implementing will only be possible given a working SCI inference implementation. The implementation on the GOS compiler is based on a formulation of SCI by Yang and Huang[5]. This SCI inference and optimisation framework is self-contained. Thus, I won't have to look at the whole compiler, rather, will only focus on understanding this optimisation framework. The inference and optimisation occurs in the following steps:

1. program source code $\xrightarrow{parse}$ untyped program
2. untyped program $\xrightarrow{convert}$ SCI program
3. SCI program $\xrightarrow{inference}$ annotated SCI program
4. annotated SCI program $\xrightarrow{optimise}$ optimised SCI program
5. optimised SCI program $\xrightarrow{convert}$ untyped program

6. untyped program $\longrightarrow$ rest of compiler

My work on the compiler will be localised to step four in the enumeration above. To apply optimisations, I need to manipulate the program tree by matching on the type annotations and free-variable interference information of the terms.

I should ensure that any added terms to the program tree have the correctly inferred SCI annotations, else, any optimisation that follows will have incorrect interference information. This makes manipulating the program tree more tedious than one might expect. There is a lot of bookkeeping involved in order to maintain correctness of annotations, meaning, inference will have to be performed while building the tree. Though many functions to build new terms are provided in the implementation, understanding the source code of the optimisation framework will be a problem. Alternatively, programs could be optimised without keeping track of SCI annotations; to sequence optimisations, the tree would have to go through the inference step again, which is inefficient. Thus, not keeping track of inference data would make reasoning about tree manipulation easier, at the expense of introducing the inefficiency of re-inferring whole programs per optimisation step. Finally, an important disadvantage of SCI is that there are no variants of it that allow active commands. This may prove to be a problem when trying to compile code that runs.

1.7 Rationale for an Optimisation

There are two reasons behind the optimisation. First is the fact that people generally write sequential programs—which is reasonable as linear structures are typically easier to consider than concurrent ones. In HLS (hardware compilation) we want fine-grained parallelisation. This is because parallelisation is cheap in hardware such as the FPGA architecture. Compared to CPU implementations of concurrency, we do not have much overhead at all, we only need to run more signals in parallel to achieve concurrent behaviour. However, concurrency is not always a safely applicable optimisation; the interference structure of the program needs to be known so we can safely use it. This is where SCI comes in, it tells us about the interference structure of a program, hence providing us the tools to describe the conditions in which said optimisation is applicable.

Secondly, the evaluation strategy used in the GOS compiler—i.e. call-by-name—can be inefficient due to potential reevaluation of arguments. It is thus an appealing goal to fix this problem, or alleviate it in some way. To do this, ground arguments—ones which satisfy the interference conditions—can be cached. This means that the optimisation should in essence change call-by-name for call-by-value for ground type arguments.

With these goals defined, the optimisation I will be designing should solve the issue of reevaluation, and do so while employing parallelisation where possible. Note, however, that the optimisation will not be equivalent to call-by-value. It is in fact much less powerful as it will not provide many of its advantages, such as closures, and will only be applicable to ground type arguments. Additionally, the optimisation is—once again—not standard. As such, optimised programs will likely appear worse out of context.

Chapter 2

Designing the Optimisation

The eventual objective is implement optimisations that employ both interference data and the ability to do parallelisation for free in hardware. These would include optimisations such as *speculative-branching* and *parallel-execution*, which are more standard optimisations. However, before tackling those optimisations, the problem of potential reevaluation should be figured out. I will be dealing with designing an algorithm to optimise reevaluation in this section. This optimisation will be very specific to the project, so it will not be standard.

2.1 Caching ground types

Given call-by-name can be potentially inefficient due to reevaluation of arguments, fixing reevaluation is an appealing goal. To alleviate the issue, we can apply an optimisation that transforms how function application terms are evaluated by caching the arguments.

The general idea is to do

$$(M\ N : Exp) \rightsquigarrow (\textbf{new}\ x\ \textbf{in}\ x := N\ ;M\ (!x))$$

to cache arguments in application terms. This effectively is call-by-value applied to ground type arguments only. Note that this optimisation is not equivalent to call-by-value—call-by-value is much more powerful. Partly, this is because the optimisation will only be applicable to ground type arguments—variables or expressions that satisfy specific interference constraints. We do not want to cache arguments if evaluating them can modify the values of other arguments or the function to which it is supplied. I will thus be looking for the following constraints in any given function argument:

- given argument is passive;
- the function we are applying does not interfere with the given argument;
- other arguments of said function do not interfere with the given argument.

From these requirements we can outline the following simple rules

$$\frac{\begin{array}{c} M : Exp \\ F : Exp \to Exp \\ (F\#M) \wedge (M \in \Pi) \end{array}}{F\ M \rightsquigarrow \textbf{new}\ x\ \textbf{in}\ x := M\ ;\ F(!x) : Exp}\ \text{Exp}$$

$$\frac{\begin{array}{c}\langle P_1, P_2\rangle : Exp \times Exp \\ F : (Exp \times Exp) \to Exp \\ (F\#P_1) \wedge (P_2\#P_1) \wedge (P_1 \in \Pi)\end{array}}{F\ \langle P_1, P_2\rangle \rightsquigarrow \textbf{new}\ x\ \textbf{in}\ x := P_1\ ;\ F\ \langle !x, P_2\rangle : Exp}\ \times_1$$

$$\frac{\begin{array}{c}\langle P_1, P_2\rangle : Exp \times Exp \\ F : (Exp \times Exp) \to Exp \\ (F\#P_2) \wedge (P_1\#P_2) \wedge (P_2 \in \Pi)\end{array}}{F\ \langle P_1, P_2\rangle \rightsquigarrow \textbf{new}\ x\ \textbf{in}\ x := P_2\ ;\ F\ \langle P_1, !x\rangle : Exp}\ \times_2$$

To briefly explain the notation, the rules are written as if they were inference rules. The conditions above the line need to be met before the transformation detailed below the line can be applied.

These rules have several problems though. First, they are not general enough. The rules only deal with application terms with an overall base type. This approach will not be able to deal with partial application terms and, as a result, will not be able to handle multiple arguments in nested application terms either.

For example, if we attempted to optimise the following

$$F\ N_1\ N_2 \qquad \text{where} \qquad F : A \to B \to C,\ \ N_1 : A,\ \ N_2 : B$$

We would only be able to potentially cache N_2 as it is the outermost layer of the nested application term. If we wanted to cache N_1, we would need another set of rules. Secondly, we would not even be able to ensure that the optimisation generates valid terms. To expand on this comment, if we naively attempted to do the following

$$\frac{\begin{array}{c}M : Exp \\ F : Exp \to Exp \to Exp \\ (F\#M) \wedge (M \in \Pi)\end{array}}{F\ M \rightsquigarrow \textbf{new}\ x\ \textbf{in}\ x := M\ ;\ F(!x) : Exp \to Exp}\ \text{Exp'}$$

the resulting program would not even type check. Sequencing ($(;) : B \to B \to B$) can only be applied to two phrases of base type B—commands or expressions. It does not make sense to sequence a command and a function, i.e. we cannot sequence a command with a partial function application. Hence, a set of more general rules are required.

2.2 Generalising with λ-lifting

If we consider the lambda-lifting process

$$\frac{\begin{array}{c}T_0 = Exp \\ B = Exp \mid Comm \\ F : T_0 \to \cdots \to T_n \to B\end{array}}{\begin{array}{c}F\ M \rightsquigarrow \\ \lambda z_1 : T_1 \ldots z_n : T_n\ .\ (((FM)z_1)\ldots z_n) \\ : T_1 \to \cdots \to T_n \to B\end{array}}\ \lambda\text{–lifting}$$

we can see that programs can be restructured in a way such that terms inside the lambda expression have a base type. This very useful as it allows us to easily generalise the previous rules to cope with partial application. To do so, we simply compose lambda-lifting with the previous rules—we lambda-lift all terms before carrying out the optimisation. It may seem inefficient to lambda-lift every single term. However, lambda-abstraction is free in hardware, so it should only affect the size of the program tree, not the actual circuit.

Composing lambda-lifting and the optimisation rule for expressions results in the following rule

$$\frac{\begin{array}{c} T_0 = Exp \\ B = Exp \mid Comm \\ M : T_0 \\ F : T_0 \to \cdots \to T_n \to B \\ (F\#M) \wedge (M \in \Pi) \end{array}}{\begin{array}{c} F\ M \rightsquigarrow \lambda z_1 : T_1 \ldots z_n : T_n. \\ \textbf{new}\ x\ \textbf{in}\ x := M\ ;\ F(!x) z_1 \ldots z_n : T_1 \to \cdots \to T_n \to B \end{array}}\ \lambda\text{–lifted Exp}$$

Similarly, lambda-lifting can be applied to the rules for products to produce generalised rules.

$$\frac{\begin{array}{c} T_{0A} \times T_{0B} = Exp \times Exp \\ B = Exp \mid Comm \\ \langle P_1, P_2 \rangle : T_{0A} \times T_{0B} \\ F : T_{0A} \times T_{0B} \to \cdots \to T_n \to B \\ (F\#P_1) \wedge (P_2\#P_1) \wedge (P_1 \in \Pi) \end{array}}{\begin{array}{c} F\ M \rightsquigarrow \\ \lambda z_1 : T_1 \ldots z_n : T_n. \\ \textbf{new}\ x\ \textbf{in}\ x := P_1\ ;\ F\ \langle !x, P_2 \rangle\ z_1 \ldots z_n \\ : T_1 \to \cdots \to T_n \to B \end{array}}\ \lambda\text{–lifted } \times 1$$

$$\frac{\begin{array}{c} T_{0A} \times T_{0B} = Exp \times Exp \\ B = Exp \mid Comm \\ \langle P_1, P_2 \rangle : T_{0A} \times T_{0B} \\ F : T_{0A} \times T_{0B} \to \cdots \to T_n \to B \\ (F\#P_2) \wedge (P_1\#P_2) \wedge (P_2 \in \Pi) \end{array}}{\begin{array}{c} F\ M \rightsquigarrow \\ \lambda z_1 : T_1 \ldots z_n : T_n. \\ \textbf{new}\ x\ \textbf{in}\ x := P_2\ ;\ F\ \langle P_1, !x \rangle\ z_1 \ldots z_n \\ : T_1 \to \cdots \to T_n \to B \end{array}}\ \lambda\text{–lifted } \times 2$$

These generalised rules would deal with all terms we want to optimise—applications where the arguments are passive and the function does not interfere with its arguments. The application is lambda-lifted, the optimisation applied, and new application terms are recursively built for every variable introduced by the lambda-lifting process.

Note that this set of rules have some nice properties:

- lambda-abstraction in Verity is free, meaning that lambda lifting can be freely applied to arbitrarily many terms without changing the cost of the resulting hardware description;
- the set of rules can be directly implemented as a recursive function, meaning we do not require mutual recursion, just a single function that checks for the constraints of each rule, and applies the optimisation.

A dummy implementation of this optimisation could be the following

$$\mathit{И}x.\ cache(M, N) = \textbf{new}\ x\ \textbf{in}\ x := N\ ; M(!x)$$

$$\mathit{И}y_{i \in \{0..n\}}.\ f(M\ N) = \begin{cases} cache(f(M), f(N)) & \text{if } M\ N : B \\ \lambda y_0 .. \lambda y_n . cache(f(M), f(N))\ y_0 .. y_n & \text{if } M\ N : T_0 \to .. \to T_n \to B \\ M\ N & \text{otherwise} \end{cases}$$

where f is the recursive optimisation function, and *cache* is a function that caches any given argument. Note that for the sake of outlining the optimisation, the term given is always assumed to pass the constraints. Moreover, the function is a non-exhaustive simplified version to save space; it is unnecessary to describe the whole algorithm. Why I deem a complete description unnecessary will be made clear when we actually run the algorithm on a simple example too see what happens:

$$
\begin{aligned}
f(F\ A\ B\ C) &= \textbf{new}\ x_0\ \textbf{in}\ x_0 := C; \\
&\quad\ f(F\ A\ B)(!x_0) \\
&= \textbf{new}\ x_0\ \textbf{in}\ x_0 := C; \\
&\quad\ (\lambda y_0.\textbf{new}\ x_1\ \textbf{in}\ x_1 := B\ ; f(F\ A)(!x_1)\ y_0)(!x_0) \\
&= \textbf{new}\ x_0\ \textbf{in}\ x_0 := C; \\
&\quad\ (\lambda y_0.\textbf{new}\ x_1\ \textbf{in}\ x_1 := B; \\
&\quad\ (\lambda y_1 \lambda y_2.\textbf{new}\ x_2\ \textbf{in}\ x_2 := A\ ; f(F)(!x_2)\ y_1\ y_2)(!x_1)\ y_0)(!x_0) \\
&= \textbf{new}\ x_0\ \textbf{in}\ x_0 := C; \\
&\quad\ (\lambda y_0.\textbf{new}\ x_1\ \textbf{in}\ x_1 := B; \\
&\quad\ (\lambda y_1 \lambda y_2.\textbf{new}\ x_2\ \textbf{in}\ x_2 := A\ ;; \\
&\quad\ (\lambda y_3 \lambda y_4 \lambda y_5.F\ y_3\ y_4\ y_5)(!x_2)\ y_1\ y_2)(!x_1)\ y_0)(!x_0)
\end{aligned}
$$

We can see from the evaluation that something is not quite right with the algorithm. It produces valid terms which are correct as far as I can tell, however, it does this in a very inefficient manner. At each step of the computation, it introduces a new lambda expression for every variable cached. From this example we can in fact observe that every row adds one more cached variable, therefore, one more lambda variable. Taking this into consideration, we can show that optimisations where all arguments pass the constraints would be of the following form

$$
\begin{aligned}
f(FN_1 ... N_n) = \textbf{new}\ x_0\ \textbf{in}\ N_n; \\
\lambda y_0.\ \cdots \\
\lambda y_1 \lambda y_2.\ \cdots \\
\lambda y_3 \lambda y_4 \lambda y_5.\ \cdots \\
\lambda y_6 \lambda y_7 \lambda y_8 \lambda y_9.\ \cdots \\
\vdots
\end{aligned}
$$

which is a triangle of lambdas n high and $n+1$ wide (an $n \times n$ staircase). We thus know that the exact number of lambdas introduced is the triangular number $(n^2+n)/2$, i.e. the sum of the first n natural numbers. Hence, the optimisation function with lambda-lifting has to iterate through the types and introduce new lambda variables at most $(n^2+n)/2$ times. This points at a complexity of quadratic order (n^2). Even though lambdas are free after compilation, during compilation we would have an added quadratic space and time growth on the number of arguments per function. This is feasible with modern day computers but not ideal. Compilation of a large enough program may be costly.

2.3 Designing a linear algorithm

In order to fix the issue with quadratic complexity, a different approach was considered. Instead of having the whole algorithm compacted into a single function, the problem was split into several components that would operate with linear complexity.

2.3.1 Types, contexts and other definitions

$$\text{type } \delta = \text{string option} \times \text{program}$$

$\Delta : \delta$ list	list of arguments to manipulate seen so far
Γ_Δ : program list	list of possibly active arguments in Δ
$\Pi : \phi$ set	set of all passive phrases
Λ : string list	list of variables needed to lambda lift seen so far

τ	some arbitrary type
$M \cap N$	M interferes with N
$M\#N$	M does not interfere with N

2.3.2 Optimisation Functions

Note that for the definitions below, extending a context always involves prepending to the list. It might seem arbitrary, but it is important to maintain consistency when splitting and rebuilding program trees. This can also be useful if we want to show correctness.

The algorithm is divided into two stages, accumulation of variables and program tree building. The first stage involves accumulating all arguments to cache and all variables needed for lambda-lifting. As the name suggests, these functions are tail recursive—we could potentially traverse very large program trees and I did not want to risk overflowing the stack.

$acc\Delta$:

$$\begin{aligned} &\forall x.\ acc\Delta\ (\Delta \mid \Gamma_\Delta \mid \Pi, N \vdash M\ N, M\#N, \forall T \in \Gamma_\Delta) \\ &\quad = acc\Delta\ (\Delta, (\text{Some } x, \text{optimise } N) \mid \Gamma_\Delta \mid \Pi, N \vdash M) \end{aligned}$$

$$\begin{aligned} &acc\Delta\ (\Delta \mid \Gamma_\Delta \mid \Pi \vdash M\ N, M \cap N \vee N \notin \Pi) \\ &\quad = acc\Delta\ (\Delta, (\text{None}, \text{optimise } N) \mid \Gamma_\Delta, N \mid \Pi \vdash M) \end{aligned}$$

$$\begin{aligned} &acc\Delta\ (\Delta \mid \Gamma_\Delta, T \mid \Pi, N \vdash M\ N, T \cap N) \\ &\quad = acc\Delta\ (\Delta, (\text{None}, \text{optimise } N) \mid \Gamma_\Delta \mid \Pi, N \vdash M) \end{aligned}$$

$$acc\Delta\ (\Delta \mid \Gamma_\Delta \mid \Pi \vdash M, M \neq M'\ N) = (\Delta\ ,\ \text{optimise } M)$$

$acc\Lambda$:

$$\forall y.\ acc\Lambda\ (\Lambda \vdash \tau \to \tau') = acc\Lambda\ (\Lambda, (y, \tau) \vdash \tau')$$

$$acc\Lambda\ (\Lambda \vdash \tau, \tau \neq \tau' \to \tau'') = \Lambda$$

The second stage involves using data accumulated in the previous stage to build the optimised program. The idea is to:

1. recursively wrap the function with all applications stored in the Δ and Λ contexts—let us call this M_1;

2. recursively wrap M_1 with all initialisations for each cached argument in Δ—let us call this M_2;

3. recursively wrap M_2 with new variable introductions for each cached argument in Δ—let us call this M_3;

4. recursively wrap M_3 with lambda abstractions for each variable in Λ—let us call this M_4;

5. output M_4 and continue traversing the rest of the tree for possible terms to optimise.

A final formalised version of the algorithm will be provided at the end of the chapter. To summarise, the effect of the transformation should be as follows:

$$(F\ M_0\ \ldots\ M_n) \rightsquigarrow (\textbf{new}\ x_0\ \textbf{in}\ \ldots \textbf{new}\ x_n\ \textbf{in}\ x_0 := M_0; \ldots; x_n := M_n\ ;\ F(!x_0)\ldots(!x_n))$$

This algorithm is linear on the number of arguments supplied to a function. It also splits up the problem into simple tasks, which will be useful if this needs to be extended. This will prove useful in the next section.

2.4 Parallelising initialisation

The algorithm described above creates new and unique identifiers for every cached argument. Moreover, we know those arguments must be passive. It is thus reasonable to suggest that we can improve the previously defined algorithm by initialising those identifiers in parallel. Note, however, that initialisation of the variables is needed before we can actually dereference them and use them as arguments. This means that the resulting program must be of form

$$\textbf{new}\ x_0\ \textbf{in}\ \ldots\ \textbf{new}\ x_n\ \textbf{in}\ (x_0 := N_0 || \ldots || x_n := N_n); F\ (!x_0)\ldots(!x_n)$$

suggesting that we can directly extend step (2) of the linear algorithm to do parallel initialisation of new variables.

2.5 Caching elements in products

Possibly the most significant drawback of the algorithm so far is the lack of rules to handle products. The functions are capable of handling products as a whole, but not individual elements in said products. This means that products can be cached if they satisfy the interference check as a whole, but would be completely ignored if they don't—even if there are elements in the product that can be cached. Since products can contain large numbers of arguments, there exists a reasonable motivation to extend the algorithm to deal with elements in products. To do this, interference constraints for elements in a product need to be defined.

In the compiler, products are constructed through nested pairs. This is good because we can define the constraints for products recursively. At any given point in a tree of nested pairs, we must check that the given pair $\langle N_0, N_1 \rangle$ satisfies the following conditions to cache N_i:

- $(N_{|i-1|})\#N_i$ the other element does not interfere with the element I want to cache;
- $F\#N_i$ the function applied does not interfere with the element I want to cache;
- $N_i \in \Pi$ the element I want to cache is passive.

We know elements in a product can only be found at the leaves of a product tree—the tree of nested pairs—thus, we know we must recursively traverse all pairs until we hit a node that is not a pair itself. At that point, we want to check for interference with every other element in the product. This is where the recursive definition of products helps us.

First property to notice is that lambda-lifting should not be modified at all. In fact, only functions dealing with caching variables—functions involving Δ—need to be extended. Next is the fact that we can more easily extend the algorithm if the type δ is updated first. We do this by adding a new constructor for products.

$$\text{type } \delta = \text{Single}(\text{string option} \times \text{program}) \mid \text{Product}(\delta \times \delta)$$

With this new type, we can define a new function to accumulate products $acc\Delta prod$.

$$\begin{aligned} &acc\Delta prod(\Gamma_\Delta \mid \Pi \vdash \langle N_1, N_2 \rangle) \\ &\quad = \text{Product}\Big(acc\Delta prod(\Gamma_\Delta, N_2 \mid \Pi \vdash N_1), acc\Delta prod(\Gamma_\Delta, N_1 \mid \Pi \vdash N_2) \Big) \end{aligned}$$

$$\begin{aligned} &\text{И}x.\ acc\Delta prod(\Gamma_\Delta \mid \Pi, N \vdash N, \forall(T \in \Gamma_\Delta).T\#N) \\ &\quad = \text{Single}(\text{ Some } x \text{ , } optimise\ N \text{ }) \end{aligned}$$

$$\begin{aligned} &acc\Delta prod(\Gamma_\Delta, T \mid \Pi \vdash N, (N \notin \Pi) \vee (T\#N)) \\ &\quad = \text{Single}(\text{ None , } optimise\ N \text{ }) \end{aligned}$$

This new function both accumulates elements to cache and applies the whole optimisation to every element in the product. Note that it uses Γ_Δ as an *explore set* of every other element in the product tree. By adding the other element of the pair at each layer, we ensure every other element is going to be checked against. This is because interference is transitive with subtrees, any given parent will interfere if any of its children interferes.

The disadvantage, of course, is that we are potentially dealing with double the number of elements per layer (2^n), and we are doing so in a non-tail-recursive way. This is not good, but there is no easy way around it if we want to consider products. Achieving tail recursion with tree traversal is not the simplest task. One could use continuations and other mechanisms to achieve this, but the base algorithm would not be readable then. The consolation is that product trees are not too deep on average, rather, they tend to be balanced by the compiler—suggesting that performance should not be too bad. There is a possibly better way of doing this, which is to check the constraints at each layer instead of the leaves. This way we could stop early, meaning we do not have to traverse the whole product tree if it is not necessary. To avoid over-complicating the description, I will be going for a simpler description of the algorithm.

To integrate function $acc\Delta prod$ to the algorithm, it must be called from $acc\Delta$.

$$\begin{aligned} &acc\Delta\ (\Delta \mid \Gamma_\Delta \mid \Pi \vdash M\ \langle N_1, N_2 \rangle) = \\ &\quad \textbf{let } N_\delta = acc\Delta prod(\Gamma_\Delta, F \mid \Pi \vdash \langle N_1, N_2 \rangle) \textbf{ in} \\ &\quad acc\Delta\ (\Delta, N_\delta \mid \Gamma_\Delta, \langle N_1, N_2 \rangle \mid \Pi \vdash M) \end{aligned}$$

Note that we are extending Γ_Δ with F when accumulating the product. This is to check interference of F with the elements of the product. Also note that $\langle N_1, N_2 \rangle$ are added to Γ_Δ for the next recursive call without checking for constraints. This is because it makes the function simpler; we don't need to check the constraints of each element in the product this way, nor do we have to potentially add several elements into Γ_Δ. This is also valid with regards to interference checking because we are being overly cautious by adding it rather than missing it—missing it if it causes interference would produce incorrect programs, checking it when it is passive is simply a wasted operation at compile-time and has no repercussion once the program is compiled into a circuit.

Next is the building functions. The easiest task here is building the application terms—we simply need to go in reverse and build the product from our type δ to a valid program tree.

$$\begin{aligned} &\mathit{build}\Delta\mathit{prod}\ (\text{Product}\ (N_{\delta 1}, N_{\delta 2})) \\ &\quad = \langle\ \mathit{build}\Delta\mathit{prod}\ (N_{\delta 1}), \mathit{build}\Delta\mathit{prod}\ (N_{\delta 2})\ \rangle \end{aligned}$$

$$\mathit{build}\Delta\mathit{prod}\ (\text{Single}\ (\text{Some}\ x\ ,\ N)) = !x$$

$$\mathit{build}\Delta\mathit{prod}\ (\text{Single}\ (\text{None}\ ,\ N)) = N$$

This is then called in $\mathit{build}\Delta\mathit{apps}$.

$$\begin{aligned} &\mathit{build}\Delta\mathit{apps}(\Delta, \text{Product}\ N_\delta \vdash M) \\ &\quad = \mathit{build}\Delta\mathit{apps}(\Delta \vdash M\ (\mathit{build}\Delta\mathit{prod}(\text{Product}\ N_\delta))) \end{aligned}$$

Initialisation is a bit more involved. My original idea was to flatten the tree and then traverse the resulting list. However, I figured that I could skip the flattening step if I simply built the tree instead of building a list.

$$\begin{aligned} &\mathit{build}\Delta\mathit{prodinit}\ (\text{Product}\ (N_{\delta 1}, N_{\delta 2})) = \\ &\quad \textbf{match}\ \Big(\mathit{build}\Delta\mathit{prodinit}(N_{\delta 1}), \mathit{build}\Delta\mathit{prodinit}(N_{\delta 2})\Big)\ \textbf{with} \\ &\quad \mid \text{Some}\ N_1\ ,\ \text{Some}\ N_2 \to \text{Some}\ (N_1\ ||\ N_2) \\ &\quad \mid \text{Some}\ N_1\ ,\ \text{None} \to \text{Some}\ N_1 \\ &\quad \mid \text{None}\ ,\ \text{Some}\ N_2 \to \text{Some}\ N_2 \\ &\quad \mid \text{None}\ ,\ \text{None} \to \text{None} \end{aligned}$$

$$\mathit{build}\Delta\mathit{prodinit}\ (\text{Single}\ (\text{Some}\ x\ ,\ N) = \text{Some}\ (x := N)$$

$$\mathit{build}\Delta\mathit{prodinit}\ (\text{Single}\ (\text{None}\ ,\ N) = \text{None}$$

To use it, the function is called from $\mathit{build}\Delta\mathit{init}$ and $\mathit{build}\Delta\mathit{inithelper}$.

$$\begin{aligned} &\mathit{build}\Delta\mathit{init}\ (\Delta, \text{Product}\ N_\delta \vdash M) = \\ &\quad \textbf{match}\ \mathit{build}\Delta\mathit{prodinit}(\text{Product}\ N_\delta)\ \textbf{with} \\ &\quad \mid \text{Some}\ X \to \mathit{build}\Delta\mathit{inithelper}\ (\Delta \vdash X; M) \\ &\quad \mid \text{None} \to \mathit{build}\Delta\mathit{init}\ (\Delta \vdash M) \end{aligned}$$

$$\begin{aligned} &\mathit{build}\Delta\mathit{inithelper}\ (\Delta, \text{Product}\ N_\delta \vdash M) = \\ &\quad \textbf{match}\ \mathit{build}\Delta\mathit{prodinit}(\text{Product}\ N_\delta)\ \textbf{with} \\ &\quad \mid \text{Some}\ X \to \mathit{build}\Delta\mathit{inithelper}\ (\Delta \vdash X || M) \\ &\quad \mid \text{None} \to \mathit{build}\Delta\mathit{inithelper}\ (\Delta \vdash M) \end{aligned}$$

A similar strategy was considered to deal with building variable introduction terms (**new** x **in** ...). However, these variable introduction terms must wrap around some other term; they cannot be sequenced or composed in parallel. Thus, I decided to sequence the function calls to the left and right of the pairs—by feeding one into the other—such that both sides of the pair are checked for variables to introduce.

$$\begin{aligned} &\mathit{build}\Delta\mathit{prodnew}\ (\text{Product}\ (N_{\delta 1}, N_{\delta 2}) \vdash M) \\ &\quad = \mathit{build}\Delta\mathit{prodnew}\ (N_{\delta 1} \vdash \mathit{build}\Delta\mathit{prodnew}\ (N_{\delta 2} \vdash M)) \end{aligned}$$

$$\mathit{build}\Delta\mathit{prodnew}\ (\text{Single}\ (\text{Some}\ x, N) \vdash M) = \textbf{new}\ x\ \textbf{in}\ M$$

$$build\Delta prodnew\ (\text{Single}\ (\text{None}, N) \vdash M) = M$$

This function is called from $build\Delta new$.

$$build\Delta new\ (\Delta, \text{Product}\ N_\delta \vdash M) =$$
$$build\Delta new\ (\Delta \vdash build\Delta prodnew\ (\text{Product}\ N_\delta \vdash M))$$

Note that the latter two building functions are tail recursive. With these additions, and by adding the "Single" constructor to the original functions, we have full support for product caching.

2.6 The refined algorithm

2.6.1 Types

$$\text{type}\ a\ \text{option} = \text{Some}\ a \mid \text{None}$$
$$\text{type}\ \delta = \text{Single}(\text{string option} \times \text{program}) \mid\ \text{Product}(\delta \times \delta)$$

$\Delta : \delta$ list	list of arguments to manipulate seen so far
Γ_Δ : program list	list of possibly active arguments in Δ
$\Pi : \phi$ set	set of all passive phrases
Λ : string list	list of variables needed to lambda lift seen so far

τ	some arbitrary type
τ_B	base types Exp, Var
$M \cap N$	M interferes with N
$M\#N$	M does not interfere with N

2.6.2 Accumulator functions

$acc\Delta$:

$$acc\Delta\ (\Delta \mid \Gamma_\Delta \mid \Pi \vdash M\ N) =$$
$$\quad \textbf{let}\ N_\delta = acc\Delta prod(\Gamma_\Delta, F \mid \Pi \vdash N)\ \textbf{in}$$
$$\quad acc\Delta\ (\Delta, N_\delta \mid \Gamma_\Delta, N \mid \Pi \vdash M)$$

$$acc\Delta\ (\Delta \mid \Gamma_\Delta \mid \Pi \vdash M, M \neq M'\ N) = (\Delta\ ,\ \text{optimise}\ M)$$

$$acc\Delta prod(\Gamma_\Delta \mid \Pi \vdash \langle N_1, N_2 \rangle)$$
$$= \text{Product}\Big(\ acc\Delta prod(\Gamma_\Delta, N_2 \mid \Pi \vdash N_1), acc\Delta prod(\Gamma_\Delta, N_1 \mid \Pi \vdash N_2)\ \Big)$$

$$И x.\ acc\Delta prod(\Gamma_\Delta \mid \Pi, N : \tau_B \vdash N, \forall (T \in \Gamma_\Delta).T\#N)$$
$$= \text{Single}(\ \text{Some}\ x\ ,\ optimise\ N\ )$$

$$acc\Delta prod(\Gamma_\Delta, T \mid \Pi \vdash N, (N \notin \Pi) \vee (T\#N))$$
$$= \text{Single}(\ \text{None}\ ,\ optimise\ N\ )$$

$acc\Lambda$:

$$И y.\ acc\Lambda\ (\Lambda \vdash \tau \to \tau') = acc\Lambda\ (\Lambda, (y, \tau) \vdash \tau')$$

$$acc\Lambda\ (\Lambda \vdash \tau, \tau \neq \tau' \to \tau'') = \Lambda$$

2.6.3 Builder functions

buildΔ*apps*:

$$\begin{aligned} &\mathit{build}\Delta\mathit{apps}(\Delta, N_\delta \vdash M : (\tau_0 \times \tau_1) \to \tau) \\ &\quad = \mathit{build}\Delta\mathit{apps}(\Delta \vdash M\ (\mathit{build}\Delta\mathit{prod}(N_\delta)) : \tau) \end{aligned}$$

$$\mathit{build}\Delta\mathit{apps}\ (- \vdash M : \tau) = M : \tau$$

$$\begin{aligned} &\mathit{build}\Delta\mathit{prod}\ (\text{Product}\ (N_{\delta 1}, N_{\delta 2})) \\ &\quad = \langle\ \mathit{build}\Delta\mathit{prod}\ (N_{\delta 1}), \mathit{build}\Delta\mathit{prod}\ (N_{\delta 2})\ \rangle \end{aligned}$$

$$\mathit{build}\Delta\mathit{prod}\ (\text{Single}\ (\text{Some}\ x\ ,\ N)) = !x$$

$$\mathit{build}\Delta\mathit{prod}\ (\text{Single}\ (\text{None}\ ,\ N)) = N$$

buildΔ*new*:

$$\begin{aligned} &\mathit{build}\Delta\mathit{new}\ (\Delta, N_\delta \vdash M) = \\ &\quad \mathit{build}\Delta\mathit{new}\ (\Delta \vdash \mathit{build}\Delta\mathit{prodnew}\ (N_\delta \vdash M)) \end{aligned}$$

$$\mathit{build}\Delta\mathit{new}\ (- \vdash M) = M$$

$$\begin{aligned} &\mathit{build}\Delta\mathit{prodnew}\ (\text{Product}\ (N_{\delta 1}, N_{\delta 2}) \vdash M) \\ &\quad = \mathit{build}\Delta\mathit{prodnew}\ (N_{\delta 1} \vdash \mathit{build}\Delta\mathit{prodnew}\ (N_{\delta 2} \vdash M)) \end{aligned}$$

$$\mathit{build}\Delta\mathit{prodnew}\ (\text{Single}\ (\text{Some}\ x, N) \vdash M) = \textbf{new}\ x\ \textbf{in}\ M$$

$$\mathit{build}\Delta\mathit{prodnew}\ (\text{Single}\ (\text{None}, N) \vdash M) = M$$

buildΔ*init* and *build*Δ*inithelper*:

$$\begin{aligned} &\mathit{build}\Delta\mathit{init}\ (\Delta, N_\delta \vdash M) = \\ &\quad \textbf{match}\ \mathit{build}\Delta\mathit{prodinit}(N_\delta)\ \textbf{with} \\ &\quad \mid \text{Some}\ X \to \mathit{build}\Delta\mathit{inithelper}\ (\Delta \vdash X; M) \\ &\quad \mid \text{None} \to \mathit{build}\Delta\mathit{init}\ (\Delta \vdash M) \end{aligned}$$

$$\mathit{build}\Delta\mathit{init}\ (- \vdash M) = M$$

$$\begin{aligned} &\mathit{build}\Delta\mathit{inithelper}\ (\Delta, N_\delta \vdash M) = \\ &\quad \textbf{match}\ \mathit{build}\Delta\mathit{prodinit}(N_\delta)\ \textbf{with} \\ &\quad \mid \text{Some}\ X \to \mathit{build}\Delta\mathit{inithelper}\ (\Delta \vdash X || M) \\ &\quad \mid \text{None} \to \mathit{build}\Delta\mathit{inithelper}\ (\Delta \vdash M) \end{aligned}$$

$$\mathit{build}\Delta\mathit{inithelper}\ (- \vdash M) = M$$

$$
\begin{aligned}
&\mathit{build}\Delta\mathit{prodinit}\ (\text{Product}\ (N_{\delta 1}, N_{\delta 2})) = \\
&\quad \textbf{match}\ \Big(\mathit{build}\Delta\mathit{prodinit}(N_{\delta 1}), \mathit{build}\Delta\mathit{prodinit}(N_{\delta 2})\Big)\ \textbf{with} \\
&\quad \mid \text{Some}\ N_1\ ,\ \text{Some}\ N_2 \rightarrow \text{Some}\ (N_1 \mid\mid N_2) \\
&\quad \mid \text{Some}\ N_1\ ,\ \text{None} \rightarrow \text{Some}\ N_1 \\
&\quad \mid \text{None}\ ,\ \text{Some}\ N_2 \rightarrow \text{Some}\ N_2 \\
&\quad \mid \text{None}\ ,\ \text{None} \rightarrow \text{None}
\end{aligned}
$$

$$
\mathit{build}\Delta\mathit{prodinit}\ (\text{Single}\ (\text{Some}\ x\ ,\ N) = \text{Some}\ (x := N)
$$

$$
\mathit{build}\Delta\mathit{prodinit}\ (\text{Single}\ (\text{None}\ \ ,\ N) = \text{None}
$$

buildΛ*apps*:

$$
\begin{aligned}
&\mathit{build}\Lambda\mathit{apps}\ (\Lambda, (y, \tau) \vdash M : \tau \rightarrow \tau') \\
&\quad = \mathit{build}\Lambda\mathit{apps}\ (\Lambda \vdash M\ y : \tau')
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{build}\Lambda\mathit{apps}\ (- \vdash M : \tau) \\
&\quad = \mathit{build}\Lambda\mathit{apps}\ (- \vdash M : \tau)
\end{aligned}
$$

buildΛ*abs*:

$$
\begin{aligned}
&\mathit{build}\Lambda\mathit{abs}\ (\Lambda, (y, \tau) \vdash M : \tau') \\
&\quad = \mathit{build}\Lambda\mathit{abs}\ (\Lambda \vdash (\lambda y.M) : \tau \rightarrow \tau')
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{build}\Lambda\mathit{abs}\ (- \vdash M : \tau) \\
&\quad = \mathit{build}\Lambda\mathit{abs}\ (- \vdash M : \tau)
\end{aligned}
$$

optimise:

$$
\begin{aligned}
&\mathit{optimise}(\Pi \vdash M\ N : \tau) = \\
&\quad \textbf{let}\ (\Delta, M') = \mathit{acc}\Delta(- \mid - \mid \Pi \vdash M\ N)\ \textbf{in} \\
&\quad \textbf{let}\ \Lambda = \mathit{acc}\Lambda(- \vdash \tau)\ \textbf{in} \\
&\quad \mathit{build}\Lambda\mathit{abs}(\mathit{rev}\ \Lambda \vdash \\
&\qquad \mathit{build}\Delta\mathit{new}(\Delta \vdash \\
&\qquad\quad \mathit{build}\Delta\mathit{init}(\Delta \vdash \\
&\qquad\qquad \mathit{build}\Lambda\mathit{apps}(\Lambda \vdash \\
&\qquad\qquad\quad \mathit{build}\Delta\mathit{apps}(\mathit{rev}\ \Delta \vdash M')))))
\end{aligned}
$$

Chapter 3

Implementation and Testing

3.1 Implementation

Implementation is a conceptually simple task; take the algorithm designed in the previous section, and implement it as it is described. The plan was to have the algorithm description at hand so I could implement it without giving the code much thought. This is, in my opinion, better than trying to work out the subtleties while coding leading to mistakes. Moreover, with a predefined algorithm, I can very simply check the code against the description for any typos or mistakes in general—which proved to be extremely useful. Of course, implementing an algorithm can be a bit more involved than following a recipe, mainly due extra work needed in code-specific problems such as manipulating the program tree.

The optimisation was implemented in OCaml, as the rest of the compiler. I will describe the steps involved in the following sections.

3.1.1 Data types and the *new* quantifier

The first task, according to the description of the algorithm, is to define types and contexts. Since we care about order, the contexts Δ and Λ are defined to be lists. This is to maintain consistency between the accumulation and building phase—programs must be taken apart and put together consistently so they are not messed up. The following are the type definitions in OCaml.

```
type 'a option = Some of 'a | None
type 'program delta  =
    Single  of (string option * 'program)
    | ProductD of ('program delta * 'program delta * 'program)
type 'program deltaD = ('program delta) list
type 'program gammaD = 'program list
type 'ty lambdaL = (string * 'ty) list
```

The next task is a definition of the *new* quantifier ($\mathcal{N}x$ for "new variable name x"). This is easily done using the imperative side of OCaml—something that completely *pure* languages cannot do as easily. First, a counter is initialised to zero and a reference of it created. New and unique names are then ensured by calling the counter and incrementing it after the call. The format used for fresh names is "0yxl__x<int>" and "0yxl__y<int>" for cached arguments and lambda lifting respectively. Names start with a zero to avoid conflicts with user-defined variables—this is illegal

syntax but valid within the compiler. The "yxl" is part of my school user name; it avoids internal name conflicts as well as making the source of the variables clear when debugging.

```
let yxl_fresh_var = ref 0

let lambda_fresh () =
  yxl_fresh_var := !yxl_fresh_var + 1;
  "0yxl__y" ^ string_of_int (!yxl_fresh_var)

let new_fresh () =
  yxl_fresh_var := !yxl_fresh_var + 1;
  "0yxl__x" ^ string_of_int (!yxl_fresh_var)
```

3.1.2 SCI operations

Before implementing any of the optimisation functions, I need to define the SCI operators for interference ($M\#N$) and passivity ($N \in \Pi$). Luckily, the SCI optimisation framework already has a function for interference, which I refactored to use. From it, a new function was defined to check interference with all elements of a given Γ context. Passivity was a bit more difficult since I had to actually define it. For it, the SCI framework contains a constraint solver; I had to solve and check the passivity constraint.

3.1.3 Accumulator functions

The accumulator functions were defined as written in the description of the algorithm. I only have to watch out for typos. The following is OCaml pseudo-code for the function signatures.

```
(*accDelta*)
let rec accD (deltalist : program deltaD)
             (gamma : program gammaD)
             ((ty, fvi, sr, prog_in) as prog : program)
             optimise (*the optimisation function*)
          : program deltaD * program) = ...

(*accLambda*)
let rec accL (lambda : 'ty lambdaL) (ty : 'ty) = ...
```

3.1.4 Builder functions

In a similar fashion to the accumulator functions, the builder functions were written by simply following the description. One difference is the way program trees are manipulated. Program trees generally have a lot of extra information: the type, free variables information, and source range of the term. This data should be kept and recomputed as I manipulate the tree so more optimisations dependent on SCI inference can be applied afterwards. However, I decided against the house keeping for two reasons:

- Housekeeping is tedious and more difficult than I expected;

- My optimisation produces invalid SCI programs anyway, so I cannot ensure the correctness of any new inference done on the output. This is because SCI cannot deal with terms of the form "`comm ; exp`", which is needed to cache arguments—this goes back to the no-active-commands problem.

Another code-specific problem were bit-widths. In the compiler we have bit-widths as types so manipulating program trees requires computing the correct bit-widths, which may require some thought. In some cases, as with dereferencing and assignment, the bit-width is straight forward; we use the dereferenced variable's or the assigned term's bit-width. In the case of initialisation, we need the bit-width of both the result and the introduced variable. For the result, we use the bit-width of the original program before optimisation; for the introduced variable, it is simply the cached argument's bit-width.

```
let getBW ((ty, fvi, sr, prog_in) as prog : program) =
  match inner_of_type ty with
  | Known(Exp (Known bw)) | Known(Var (Known bw)) -> bw
  | _ -> 0
```

Following the description, builder functions are divided into two categories: argument caching and lambda lifting. They are easy to tell apart by the use of "D" (delta) or "L" (lambda) in their names. To save space, the functions won't be defined here. They can be found in the code.

3.1.5 The optimisation function and integration

All of the functions defined are then brought together by a single function. This function just chains all the builder functions and traverses the tree to pattern-match for application terms.

```
(*Call-by-value on passive and non-interfering arguments*)
let rec cbv_ground_types ((ty, fvi, sr, prog_in) as m : program)
        :program =
  let make_prog term = (ty, fvi, sr, term) in
  let optimise = cbv_ground_types in
  match prog_in with
  | Freevar s -> m
  | Lambda (s, inf , prog) -> make_prog (Lambda(s,inf,optimise prog))
  ...
  | Apply (prog_1 , prog_2) ->
     let (deltalist, m') = accD [] [] m optimise in
     let lambda   = accL [] ty in
     let napps    = buildDapps (List.rev deltalist) m' in
     let lapps    = buildLapps lambda napps in
     let dinit    = buildDinit deltalist lapps m in
     let dnew     = buildDnew deltalist dinit m in
     let newprog  = buildLabs (List.rev lambda) dnew in
     newprog
```

With this defined, the optimisation is implemented. The only step left is integration of this optimisation with the existing SCI optimiser. To do this, I had to add my optimisation at the end of the optimisation chain. This can be seen in the `new_optimizer` function, which was provided with the framework.

3.2 Testing

To test for validity of the output, I compiled a few sample programs and manually verified that the output is correct. Ideally, I would need a formal proof of correctness for the algorithm, however, that is beyond the scope of this report. For benchmarking, I installed VHDL simulation and project building software ModelSim and Quartus2 so I could check the resources and run time of the optimised programs. Unfortunately, the lack of support for active commands meant that none of the programs are executable. I tried to run the programs through a main, but was unsuccessful. This means that no testing could be done where execution is required.

3.2.1 Sample programs to Optimise

These are the programs I used. They have been defined to test specific parts of the compiler. The programs are small because the resulting program tree will have to be manually checked for correctness.

Sample program 1 (SP1) tests whether the optimisation can cache and argument, and then optimise the cached argument by recursively traversing it.

```
(*SP1*) let f = (\y . y + y) ((\x.x+1) 2) in export f
```

Sample program 2 (SP2) tests whether the optimisation can handle multiple parallel initialisation and recursive traversal of product terms.

```
(*SP2*) let f = (\(a,b,c,d,e).a+b-c+d-e) (1,2,3,4,5) in export f
```

Sample program 3 (SP3) tests whether the optimisation can handle partial application. The algorithm needs to cache the existing arguments and put the tree together correctly.

```
(*SP3*) let f = (\y.\z.\w . y + z + w) 2 in export f
```

Unfortunately, it seems the SCI framework has bugs. Thus, I will not be using it for the tests. Instead, I'm going to force true or false outputs from the SCI functions to test the code. This is a shame since I had spent a lot of time learning to use the framework and integrating my optimisation to it.

3.2.2 Correctness test with forced active terms

With this test, I wanted to verify that the optimisation is always putting the programs together correctly. To do this, I changed the passivity function to always return false; i.e. all terms are active.

```
(*Optimised SP1 with forced active terms*)
let f = (\y . y + y) ((\x.x+1) 2) in export f
```

Result: Optimiser has left the program unchanged; it is consistently taking apart and putting the program back together without making mistakes.

```
(*Optimised SP2 with forced active terms*)
let f = (\(a,b,c,d,e).a+b-c+d-e) (1,2,3,4,5) in export f
```

Result: Optimiser has left the program unchanged; again, it is consistently taking apart and putting the program back together without making mistakes.

```
(*Optimised SP3 with forced active terms*)
let f = (\y2. \y1. ((\y.\z.\w . y + z + w) 2) y2 y1) in export f
```

Result: Optimiser has not left the program unchanged. No arguments were cached, which is correct behaviour. Only transformation applied is lambda-lifting, which is also correct behaviour. All partial applications are lambda-lifted because lambda abstraction is free in the compiler.

3.2.3 Correctness test with forced passive terms

This aims to test whether arguments are being correctly cached. All terms are passive; both interference and passivity functions return true. Sample programs are small because optimised program trees can be quite big.

```
(*Optimised SP1*)
new x7 in
(x7 := new x4 in
 (x4 := new x1 in
  (x1 := 2) ;
  (\x. new x3 in
   new x2 in
   (x3 := x || x2 := 1) ;
   ((!x3) + (!x2))) (!x1)) ;
 (\y. new x6 in
  new x5 in
  (x6 := y || x5 := y);
  ((!x6) + (!x5))) (!x4)) ;
(let f = (!x7) in
 new x8 in (x8 := f) ; (export (!x8)))
```

Result: The tree is much larger, but all extra operations are cheap if not free, plus they eliminate reevaluation. The optimisation correctly caches every argument; recursively traversing and optimising arguments, as well as initialising new variables in parallel.

```
(*Optimised SP2*)
new x14 in
(x14 :=
   (new x5 in new x4 in new x3 in new x2 in new x1 in
    (x5:=1 || x4:=2 || x3:=3 || x2:=4 || x1:=5) ;
    (\a.\b.\c.\d.\e.
     new x13 in new x6 in
```

```
     (x13 := (new x12 in new x7 in
              (x12 := (new x11 in new x8 in
                       (x11 := (new x10 in new x9 in
                                (x10 := a || x9 := b) ;
                                ((!x9) + (!10)))
                               || x8 := c);
                       ((!x11)-(!x8)))
                      || x7 := d) ;
              ((!x12) + (!x7)))
             || x6 := e);
     ((!x13)-(!x6)))
      (!x5) (!x4) (!x3) (!x2) (!x1))) ;
(let f = (!x14) in
 new x15 in (x15 := f) ; (export (!x15)))
```

Result: Optimiser has correctly traversed cached all elements of the product. Operations here are parallelised, even if they are not balanced. This is good because products are normally dealt with sequentially. This shows how initialising in parallel has the added advantage of parallelising operations. Normally, the tuple lookup time for each element is added. After the optimisaion, we only wait for the longest lookup time since all of them are composed in parallel.

```
(*Optimised SP3*)
let f =
  (\y7.\y6. new x1 in
   (x1 := 2);
   (\y.\z.\w. new x5 in new x2 in
    (x5 := (new x4 in new x3 in
            (x4 := y || x3 := z);
            ((!x4)+(!x3)))
           || x2 := w);
    ((!x5)+(!x2))) (!x1) y7 y6)
in export f
```

Result: Optimiser has correctly cached the argument 2 in $x1$. Lambd lifting is also correctly done; $y7$ and $y6$ are applied in the correct order which means the algorithm is being consistent. Moreover, addtion has been parallelised.

3.2.4 Benchmarking

The plan here was to compile two versions of the three test programs—with and without the optimisation—and use Altera Quartus II and ModelSim to benchmark the resulting VHDL code. Benchmarking would occur in two stages: DE2 project building using Quartus 2 to view resource usage and other compilation reports; and simulation using ModelSim. Unfortunately, due to some theoretical limitations of the SCI inference system, it was not immediately possible to benchmark the code. The limitations include:

- current SCI infernce implementation cannot handle libraries, so I could not use libraries like "<print>";
- current SCI frameworks cannot handle expressions of the form "comm;exp", which means I cannot compile terms such as "new x in x := 2 ; x";

- the compiler uses libraries for all communication with hardware, so I cannot produce code that communicates with hardware.

Due to a combination of the previous factors and other limitations, I was unable to compile programs that could be run directly; hence the use of "export" in the sample programs. On the plus side, I know that optimised code should almost be faster by construction. I will discuss this in the next section.

Chapter 4

Algorithm Evaluation and Discussion

4.1 Correctness

From the correctness test I can tell that the algorithm appears to be generating correct terms. All programs generated so far are valid, and manual checking shows that they all have been optimised correctly. This, however, does not really prove my algorithm is correct; it only shows that it appears correct so far. Since I cannot test my algorithm on all possible programs, I would need a proof of correctness. To do this, I would need to define correctness first, which I take to be extensional equality between the input and output programs. Unfortunately, formally proving that my algorithm is correct would be quite complicated and beyond the scope of the project. Informally, however, the algorithm should be correct because the algorithm is taking apart programs and putting them back together in a consistent manner. Given I'm using the same components of a given program to build the optimised version, as long as I'm not putting it together in a different order, it should remain the same program extensionally. Moreover, the transformations involved in the optimisation cannot change extensionality of input programs; caching variables only changes the evaluation strategy while lambda-lifting produces extensionally equal terms by definition. This suggests that applying the transformations while being consistent should not change extensionality of optimised programs.

Assuming the algorithm is correct—which it seems to be—the optimisation would be caching variables and initialising them in parallel. Parallel initialisation is especially useful as it also performs parallelisation of sequential operations cached. This means that the algorithm not only fixes reevaluation of ground type arguments, but also improves the performance of cached sequential operations by initialising them in parallel. If these initialisations are built with balanced trees, it might be possible to achieve tree-like parallel reduction of nested operations.

4.2 Benchmarking the Optimisation

Though I was unable to execute optimised programs, it is still reasonable to ask how much better the optimised code is compared to the non-optimised program. Intuitively, the algorithm should be producing more efficient code by construction. However, the program trees are very large and they appear to be pesimisations if anything. To answer the question, the complexity of the programs can be formalised. It can be shown that the complexity of the original programs in the

form of a single-argument application is:

$$|F\ M| = |F| + K_F \cdot |M| \qquad \text{where } K_F \in \mathbb{N}$$

K_F is a constant that depends on the function F; it is based on how many times the argument occurs in F, i.e. how many times M has to be reevaluated. Contrastingly, the optimised program would have the following complexity:

$$|\textbf{new}\ x\ \textbf{in}\ x := M\ ; F\ (!x)| = |F| + |M| + 1$$

Note that this already shows that the optimised program has the potential to be much more efficient than the original. However, it should be noted that a constant number of clock cycles (the $+1$) is used to set up the infrastructure for caching. This means that in the event that M is an extremely low cost term—e.g. a constant with cost equal to 1—then the optimisation would not produce better performing code. The result would be equal if not slower by a cycle. It should be noted however that my optimisation is applied after standard optimisations such as *constant folding* and *constant propagation*. These optimisations should get rid of the scenarios where my optimisation is just caching constants, which is the worst-case scenario.

The significance of the optimisation becomes much clearer when several function arguments are involved. For a function application with two arguments, we can show that the *cost* would be:

$$|F\ M\ N| = |F| + K_F \cdot |M| + K_F \cdot |N|$$

Compare to it, the optimised version would be as follows:

$$|\textbf{new}\ x, y\ \textbf{in}\ (x := M || y := N); F(!x)(!y)| = |F| + \max(|M|, |N|) + 2$$

Now the difference is potentially much higher. Instead of a sum of two factors, we have a single maximum factor. So the addition of clock cycles wasted in setting up the infrastructure is even less meaningful in comparison to the performance gained from parallelising initialisation. In a bad case scenario—if the cost is 1—the optimisation is still able to be much better, i.e. unoptimised $|F| + 2 \cdot K_F$ compared to optimised $|F| + 3$. If F is using both arguments at least once, then we have cost of unoptimised $|F| + 4$ compared to $|F| + 3$. In the scenario where the arguments are not used, previous optimisations should optimise the arguments out. This specific scenario with functions not using their arguments, however, is one of the rare cases where call-by-name is actually better than call-by-value; the argument isn't even evaluated in first place. This means that my optimisation would be meaningless in this case as it would only add overhead.

Finally we consider an arbitrary number of arguments. This is the scenario where the optimisation is able to do much better; the more arguments the more potentially better the optimised code it produces. Consider the unoptimised cost first:

$$\left|F\ \vec{M}\right| = |F| + \sum_{i=1}^{n} K_F \cdot |M_i|$$

Now consider the optimised cost:

$$\begin{aligned}
&|\textbf{new}\ \vec{x}\ \textbf{in}\ (x_1 := M_1 || \ldots || x_n := M_n); F(!\vec{x})| \\
&= |F| + \max\left|\vec{M}\right| + c \\
&\qquad \text{where} \\
&\qquad c = \log n \quad \text{for balanced initialisation trees} \\
&\qquad c = n \quad \text{for unbalanced initialisation trees}
\end{aligned}$$

Given a cost of $|M| = 1$ to consider, the cost would go from $|F| + n \cdot K_F$ to $|F| + 1 + c$, where c is either $\log n$ or n. This means that in the bad case where $K_F = 1$, the cost become $|F| + n$ unoptimised and $|F| + 1 + n$ or $|F| + 1 + (\log n)$ optimised with unbalanced and balanced initialisation

respectively. From this we can observe that even on a bad case—which would be optimised out before this optimisation runs anyway—the cost can be equal or even lower than the unoptimised version. This suggests that the optimisation is—by construction—very close if not equal in a bad case and much better in an average to good case. However, this also shows a problem with my algorithm; I only build initialisation trees by linearly wrapping them around recursively. This means that there is room for improvement; I could build balanced initialisation trees instead, which would ensure that optimised code with arbitrary arguments would always be better, even in the worst cases.

4.3 Performance

In terms of computational cost, the algorithm is linear with mostly linear tail-recursive components. This is to keep compile-cost low; keep memory usage low by reusing the same stack-frames with tail recursion, and keep time taken linear. This is much better than the original naive solution which ran in quadratic complexity. This difference is significant considering that the functions which deal with products are not tail-recursive. This is not ideal and the performance difference might have been noticeable if the naive version were implemented with product traversal. On the plus side, products should not be very deep on average as they are balanced by the compiler. This means that traversing products should not be as computationally expensive as whole program traversals on the average case, which make its possible bad performance less meaningful in comparison. This is, however, a place where there is room for improvement.

4.4 Limitations of SCI

There exists a problem which relates to a limitation of SCI formulations. I had to test the optimisation with programs that could not have commands sequenced with expressions. This is because no SCI framework allows the construction of "`comm; exp`" terms in programs; no active commands. This unfortunately means that programs that result from my optimisation are not correct SCI programs; which also implies that optimisations that depend on SCI inference cannot be sequenced after my optimisation. This suggests that my optimisation must always come at the end of a block of SCI dependent transformations. There could be exceptions or ways to make a data structure such that SCI annotations are still valid for incorrect SCI programs, however, without formalisation of the properties, I have no way of checking the validity of this proposal. Ideally, a formulation of SCI which allows active commands is needed.

Chapter 5

Project Management

5.1 Evaluating Project Planning

The original plan to address the project was to:

1. familiarise myself with SCI and all the papers involved;
2. familiarise myself with the SCI inference framework;
3. design the algorithm to cache ground-type arguments;
4. implement the algorithm;
5. work toward speculative branching.

Unfortunately, most steps were underestimated in time taken and difficulty; step 2, 3 and 4 are the main culprits. Step 2 took much longer than I expected, and step 3 and 4 were repeated a lot of times. Overall this meant that I had to scrap addressing speculative branching.

First we consider step 2; the SCI inference framework is important to the project as I had to ideally fit my optimisation into it. Thus, a lot of time was spent learning about the framework. This involved learning about the author's coding style and idiosyncrasies. The coding style was particularly difficult to grasp due to the numerous levels of indirection: the use of parametrised data structures made it difficult to work out the program types; numerous domain-specific-languages are used to interact with the trees; the code has many monadic operations that are used throughout the framework; and no comments were present in the source code. The framework was documented, but it was either not always clear or the description was too high-level. Fortunately, the author was available to me as he is a student of the university. However, too many weeks were spent trying to work out meeting times and pestering him to get explanations.

For steps 3 and 4, though not detailed in the design section, I implemented several versions of the optimisation only to discover they were either not exhaustive (e.g. handling partial application) or inefficient (e.g. quadratic complexity vs. linear). This means that I would spend a lot of time writing code to then scrap all the work because I had thought of a better way of doing things. It is likely that I might have found inefficiencies or noticed improvements by implementing the optimisations anyway—i.e. the thought process involved in implementing code might have helped me notice things about the algorithm—however, I would still consider this poor project management. If I had worked out the problems before hand, I could have saved a lot of time. I must state, however, that this is only with hindsight—it is maybe the case that I should not realistically expect to work out all the problems right at the start, especially those involving implementation.

Another issue is a problem involved in using code that someone else wrote without maybe properly checking for its correctness. Near the end of the implementation phase, I found out that the SCI framework had some bugs in it; infinite loops and problems with the SCI constraint operators. This was more time spent contacting the author and trying to find out what is wrong instead of testing my optimisation or benchmarking it. I had to consideration at one point writing my own isolated dummy-framework just to test the algorithm, which makes learning about the original framework even more wasteful. It is not unreasonable to expect the repetitions of steps 3 and 4 in a software-engineering project; they would effectively form cycles of design and improvement. My mistake is the fact I didn't evaluate my naive designs before implementation; which is wasteful but might have helped me notice the problems.

There were cases, however, where project planning worked out; not everything was wasteful. For instance, as the project progressed, I picked up on the inefficiencies of my plan, and fixed them. The last iteration was the project as I would have ideally wanted it in first place: design an algorithm, formalise it, implement it from the formalised description, test it, improve it. Writing a formal description of the algorithm was especially important since it allowed me to painlessly implement the algorithm and then verify it. The optimisation ran correctly as long as the implementation closely followed the description. This means that I was able to use the description to find any bugs in my code; by reading the description and matching it line-by-line with the implementation, it becomes easy to spot mistakes. Additionally, I followed a strict rule to always document all my work, which made writing this report a lot easier. This is particularly important when we consider multiple clashing deadlines for several modules near end of term. However, more importantly, documenting all the algorithms designs I had thought about has allowed me to contrast them and evaluate my decisions.

Overall, I am satisfied with the outcome of my project; I do recognise that my caching ground-type arguments algorithm can be considered a realistic goal. The original idea was, however, to implement an optimisation to cache ground-type arguments and then move on to speculative branching. A combination of time constraints, underestimating the difficulty of the task, and implementation issues prevented this. That said, I am satisfied with the optimisation I designed; the algorithm is a bit more complicated than I like, but works with linear complexity. This is good because not all operations carried out by the optimisation are linear; constraint solving is generally not linear for instance. Given that the optimisation is not as trivial as I had originally thought it would be, designing and implementing it is a realistic goal.

Moreover, again with hindsight, the next potential optimisation—speculative branching—would be much more involved than caching ground-type arguments. With this optimisation, we have three signals running in parallel: the condition, the first statement (true), and the second statement (false). This suggests that I would not be able to limit my work to the front end of the compiler; the optimisation requires a way for two "wires" to signal each other after the condition is evaluated, which cannot be done through mere transformations of the program tree. Given more time, it is possible I could have tackled speculative branching. However, seeing it is a much more complex task than the one I achieved with this project, it seems unlikely.

5.2 Improvements and Further Analysis

Given more time I would have implemented the improvements detailed in the algorithm evaluation; first of which is building balanced initialisation trees. My first approach to a linear algorithm was too simplistic and I had not considered parallel initialisation straight away. I only noticed the possibility of balanced initialisation trees after the optimised cost was formalised. Another possible improvement is to write a tail-recursive product traversal. This can be a bit subtle and might make the algorithm less readable, so it might not be the best improvement.

As for the original plan, after considering the time taken to finish this project, speculative-branching

might have been an unrealistic goal. If I were to do this project again, I would have planned better and evaluated my designs before implementing them, and not bothered with all the effort getting my algorithm integrated with the SCI framework since it might have some bugs and SCI does not even let me compile code that runs. At one point I was unnecessarily worried about integration of my optimisation with the rest of the project; it is needed for testing, but getting it designed and implemented comes before integration.

In addition, there is no formal proof of correctness for the algorithm. The informal and intuitive correctness has been discussed, but a formal proof is ideally desired. A formal proof could be a lot more involved, and due to time constraints and difficulty, is not attempted in this project. This leaves a proof of correctness as possible future work, which I would have attempted if given more time.

Chapter 6

Conclusion

In this report I have designed and implemented an optimisation to approach the problem of reevaluation with call-by-name whilst using the benefit of parallelisation in FPGA hardware architectures. I did so by presenting the design process, which involved designing incomplete, generalised naive, and generalised linear algorithms to solve the problem. Moreover, I provided sample test programs and an evaluation of the correctness of the optimisation, a benchmark of optimised programs, and the algorithm performance. As for the effectiveness of the algorithm, I showed that optimised code is on average cheaper than unoptimised code. In the worst cases, it is currently a bit more costly, if not equal. However, this is where improvement would have its impact as it can guarantee that the algorithm always produces cheaper code, even in the worst cases, for arbitrary numbers of arguments. The division of work into simple components makes the algorithm easily extensible. This is of particular importance because there is still room for improvement. Such improvements involve balanced initialisation trees and tail-recursive product traversal. Both of these are easily implemented by modifying the related component in the algorithm. Balanced initialisation trees can be achieved by changing the way that the initialisation-tree builder function puts the tree together, while tail-recursive product traversal simply requires replacing the product accumulation function.

From this I can conclude that the optimisation problem was dealt with successfully and efficiently through a non-standard linear algorithm with mostly tail-recursive components. It is important to note that the non-standard optimisation is very specific to the problem, and might not be useful out of context. With CPU implementations of parallelisation, or with languages that do not use call-by-name evaluation, the optimisation detailed here would in fact hinder performance. Finally, I only discussed the informal and intuitive correctness of the algorithm, thus formally proving the correctness of the algorithm is left as possible future work.

Bibliography

[1] Dan R. Ghica, Alex I. Smith, and Satnam Singh. Geometry of synthesis iv: compiling affine recursion into static hardware. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 221–233. ACM, 2011.

[2] Peter W. O'Hearn, John Power, Makoto Takeyama, and Robert D. Tennent. Syntactic control of interference revisited. *Theor. Comput. Sci.*, 228(1-2):211–252, 1999.

[3] John C. Reynolds. Syntactic control of interference. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 39–46. ACM Press, 1978.

[4] John C. Reynolds. Syntactic control of inference, part 2. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer, 1989.

[5] Hongseok Yang and Howard Huang. Type reconstruction for syntactic control of interference, part 2. In *ICCL*, pages 164–173, 1998.

Appendix A

Appendix

This project report has been submitted with a compressed file (lin_1228863.zip) which contains a directory called `compiler`. In this directory, the file under `src/sci_opt_optimizer.ml` contains all the code I wrote for the project.

To compile the compiler, while in the `compiler` directory run `make`. This requires OCaml and will require a UNIX-like environment; e.g. Cygwin if using Windows.

To compile any Verity code using the compiler, run `./gosc` on the target file with the correct options. For more information visit: http://www.veritygos.org/documentation.

My optimisation is active by default. To disable it, comment out line 59 from `src/gosc.ml`:

```
optimizations := StringSet.add "cache-ground-types"    !optimizations;
```