Department of Computer Science

Technical Report

Undergraduate Dissertation: Compilation and Automatic Parallelisation of Functional Code for Data-Parallel Architectures

Tommy Lillehagen

Technical Report 2011-01 **July 2011**

ISSN 1740-9497

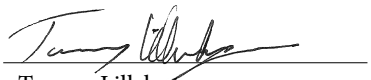
Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Compilation and Automatic Parallelisation of Functional Code for Data-Parallel Architectures

Tommy Lillehagen
B.Sc. (Hons) Computer Science
University of Bath

May 2011



Tommy Lillehagen

Compilation and Automatic Parallelisation of Functional Code for Data-Parallel Architectures

Submitted by: Tommy Lillehagen

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Tommy Lillehagen

Abstract

Over recent years, there has been a stagnation of the increase in CPU clock speed, and consequently, it has become increasingly popular to offload general-purpose computing problems to graphics processors to try to exploit the massively data-parallel processing capabilities of these devices.

This project presents the design of a functional programming language and the implementation of a prototype compiler which aims to produce code that exploits the powerful processing capabilities of data-parallel hardware components, such as CUDA-enabled graphics processors. One of the long-term goals is to provide programmers with a tool that simplifies the development of algorithms for parallel architectures.

Previous work in the area of automatic parallelisation of code is predominantly concerned with the exploitation of task parallelism in functional languages, such as Lisp and Haskell, and data parallelism in imperative languages, such as Fortran. In the cases where data-parallelism has been exploited in functional languages, *e.g.*, in Data Parallel Haskell, this has mostly been done by introducing library support for CUDA, OpenCL and other data-parallel frameworks.

The main focus in the course of this project has been directed towards the optimisation techniques that can be applied to seemingly sequential, functional-style code to prepare it for automatic parallelisation. The preeminent transformation in this context is the conversion of augmenting recursion and tail recursion into iteration which, consequently, can enable the translation of iterative constructs into parallel loops, given that there are no loop-carried dependences.

The compiler strives to identify natural mapping and reduction constructs in sequential code. Furthermore, a dynamic performance model is employed to ensure that only beneficial sections of the code are parallelised. It is concluded from the initial results, that tenfold to hundredfold speedups can be achieved from the parallelisation of sequential representations of naturally data-parallel constructs, depending on the point of comparison.

Contents

Algorithms

Figures

Listings

Tables

Acknowledgments

First, I would like to express my sincere gratitude to my supervisor Dr. John Power for his continued help and support, and to Dr. Russell Bradford and Prof. John Fitch for their input and advice in the early stages of the project. I would also like to thank David Anisi, Peji Joe Faghihi, Phillip Meredith and Martin Shaw for proof-reading and support. Last but not least, I would like to thank Marius Eriksrud and Robin Helgesen for providing me with the hardware equipment that was used in the performance testing of the project.

CHAPTER 1

Introduction

Contents

1.1 Background

Parallelism was explored already in the early days of computing, and today, the approaches to parallel computing are many. On a conceptual level, parallelism is generally classified into two groups, namely task parallelism and data parallelism. Task parallelism is the term used when we execute different sequences of instructions simultaneously. Each of these code sequences can access both shared and local data. Conversely, data parallelism is the term used when we execute a single sequence of instructions on multiple data-sets[1] [SK10, §10].

On a hardware level, we find parallelism realised in the form of, for instance, instruction pipelining, hyper-threading and SIMD[1] within a register (SWAR). All these techniques allow the user to speed up computations by concurrent or seemingly concurrent execution of instructions.

1.2 Motivation

Over recent years, the rate of increase in clock speed of CPUs has been flattening, contributing to a boost in the popularity of parallelism [Sut05]. The development of the uni-core CPUs has stagnated, and engineers are now working to increase the number of cores per CPU. Work is also put into improving the infrastructure between these cores as the achievable memory bandwidth is a limiting factor. Today, multi-core CPUs and

[1] SIMD – single instruction, multiple data

other hardware that can be exploited to attain concurrency have become mainstream and can be found in almost every new laptop and PC.

The graphics processing unit (GPU), a coprocessor found on most computers nowadays, was originally designed to offload 2D and 3D graphics processing from the CPU. Dedicated graphics processing chips have existed for many years, but in recent years it has become increasingly popular to utilise the GPU's massively data-parallel processing capabilities in general-purpose computing (GPGPU) [SK10, §1]. In 2005, NVIDIA published work on how computational concepts can be mapped onto the GPU [Har05]. At the same time, they also worked on unifying their GPU architectures. The result, known as the Compute-Unified Device Architecture (CUDA), simplifies GPGPU programming and allows the programmer to easily utilise the GPU in the computation of data-parallel problems. The first version of the CUDA Toolkit, a framework consisting of a specialised C/C++ compiler, a profiler and various other tools, libraries and documentation for programming CUDA-enabled devices, was released in 2007 [NVI09].

Even though hardware is in place to perform parallel computations, the exploitation of these features is not synonymous with higher performance. Not all computational problems can be implemented efficiently on parallel architectures, and to rewrite the algorithms to add support for parallelism can be a nontrivial task. Furthermore, the complicated nature of concurrent problems, the cost of transferring data between memory banks, *e.g.*, for GPUs, and the communication overhead that is existent between co-operating threads and processes all place a limitation on the amount of speedup that is achievable from parallelisation [Har05, p. 494].

The motivation for this project originates in the problems that are stated above. The problem of parallelising existing sequential algorithms constitute a challenge to the programmer, and the emerging use of parallelism in general makes it important to be able to express parallel algorithms in a clear and concise way.

1.3 Aims

The overall aim of this work is to provide the programmer with a tool that simplifies the exploitation of the processing capabilities offered by today's data-parallel architectures. More specifically, this work targets the NVIDIA CUDA chipset, which is an apt representative for these architectures.

Previous research indicates that functional programming languages apply well to concurrent applications [AVWW93, Arm07, Jon89]. Moreover, the functional programming paradigm have a good reputation in the area of correctness analysis since the functional programming style enables and simplifies mathematical reasoning about programs. Consequently, a functional language is chosen as the foundation for the work carried out in this project.

The work that is described in this document entails the development of a prototype compiler for a functional language, namely the language which is described in Appendix A. The objective of this compiler is to generate optimised code which exploits the processing capabilities of the GPUs. The focus is directed towards the optimisation techniques that can be applied to functional code to prepare the input for automatic parallelisation, *i.e.*, on the steps that can be taken to simplify the semantic analysis of the input programs.

Task parallelism is outside the scope of this project due to the restrictive nature of CUDA. The implemented compiler solely targets flat data-parallel architectures. Hence, no nested parallelism is available to the end-user. Data-parallel architectures maintain

a strong relation to *mapping* problems, *e.g.*, the original applications of GPUs were essentially mapping operations that were carried out on two-dimensional arrays of pixels. Mapping is also a renowned phenomenon in the context of functional programming, and the prototype compiler capitalises on this link during the compilation process when parallelising the input code.

Functional programming languages motivate the programmer to express problems using recursive functions. Hence, the occurrence of recursion and function pointers plays a vital role in conventional functional languages. CUDA does not support any of these two language features prior to devices with compute capability[2] 2.0. Hence, an alternative approach must be taken when targeting these devices [NVI10b]. A part of the problem can be met by incorporating optimisation techniques such as tail-call elimination and by analysis of the structure of the language [ALSU06]. Furthermore, data-parallel recursion can be used to iterate over recursively defined data [NO99].

It is reasonable to assume that there are situations that cannot be handled efficiently in this manner. Thus, an important job of the compiler is to determine what can be parallelised and, hence, executed on the GPU. It is noteworthy that the lack of architectural support for recursion and function pointers also applies to other data-parallel architectures such as, for instance, OpenCL [Mun10].

1.4 Objectives

The following bullet points list the quantitative and qualitative measures by which the completion of this project is judged:

- A *statically* and *implicitly typed*, functional programming language must be designed. The language must allow the programmer to easily express algorithms in a *functional* manner.
  - There are penalties associated with type conversion. These are particularly high on delicate architectures such as CUDA. *Implicit typing* allows the compiler to deduce an optimal type system by minimising the number of type conversions in the program.
  - *Static typing* reduces the memory overhead that is involved in the abstraction of typing, and the performance overhead that is involved in the boxing and unboxing of dynamically typed values.
- An *optimising* and *parallelising* compiler must be implemented. The compiler must support the programmer in the development of high-performance programs that can be executed on data-parallel architectures. More specifically, compilation must yield programs that, if possible, can exploit the massively data-parallel processing capabilities of CUDA-enabled graphics processors.
  - Automatic parallelisation of code is heavily dependent on an extensive analysis of the input program. To be able to successfully conduct such an analysis, the input should be well-formed and optimised. Thus, much attention should be drawn towards the general *optimisation techniques* that can be applied to functional code, in the belief that this will improve the quality of the parallelisation process.

[2]The term "compute capability" is used by NVIDIA to discriminate between the feature-sets of their CUDA-enabled graphics processors.

1.5 Document Structure

This document is structured into seven chapters. The first chapter, as we have already seen, discusses the background, the motivation and the overall aims and objectives of the project. The second chapter is the result of the literature survey that has been undertaken to get a better understanding of the topic area, and aims to give a comprehensive overview of the problem of automatically parallelising sequential, functional-style code.

Chapters 3 to 5 present the requirements, design and implementation of the suggested solution. More specifically, Chapter 3 lists the identified functional and non-functional requirements, in addition to a set of testing requirements. Chapter 4 presents the high-level design of the implemented compiler system and draws attention to the overall objectives of the compiler. Chapter 5 discusses the details around the low-level design and implementation, and justifies the techniques that have been applied and the choices that have been made during the development phase.

Chapter 6 describes the organisation and the results of the unit and black-box testing. This chapter also presents the results of the performance testing, which contains objective measurements of the speedups that have been achieved as a result of the employed optimisation and parallelisation techniques.

In Chapter 7, we summarise the work and critically appraise the implementation and the findings. In this chapter, we also identify areas for further work.

The chapters that are mentioned above constitute the main matter of this document. There is also a set of appendices which expand on the technical content that is presented in Chapters 4 and 5.

Appendix A gives a formal description of the functional programming language that has been designed as part of this work, and Appendix B lists and justifies the algorithms that have been implemented to convert instances of augmenting recursion into tail recursion. In Appendix C we define the type and performance models that are used by the implemented compiler system, and in Appendix D we give an overview of the command-line usage of the compiler and a brief introduction to the source code hierarchy. Appendix D also lists a set of representable code excerpts.

CHAPTER 2

Literature Review

Contents

2.1 Introduction

Over recent years, the rate of increase in CPU clock speed has been flattening, contributing to a boost in the popularity of parallelism [Sut05]. Dedicated graphics processors have delivered accelerated performance to end-users for years, but not until recently has it become popular to utilise the capabilities of these parallel devices to do more than

just graphics processing [SK10, §1.3]. GPUs are designed to carry out massively data-parallel operations such as pixel shading and, thus, allow the user to run the same sequence of instructions on multiple data-sets simultaneously. Because of this, GPUs are today frequently used in data-intensive computing, especially in domains such as medical imaging, computational fluid dynamics and environmental science [SK10, §1.5].

The trend of using high performance graphics processors to perform general-purpose computing was for a long time deemed a challenging business. Traditionally, the resources on these devices were tightly constrained and the instruction set was limited to specific graphics operations only [SK10, §1.3.2]. However, in later years, GPU manufacturers such as NVIDIA and AMD/ATI have put efforts into unifying the hardware architectures of their graphics platforms to provide a simple and generic programming model for data-parallel computing. Some of the results of these initiatives are NVIDIA's Compute Unified Device Architecture (CUDA), Khronos Group's Open Computing Language (OpenCL), Microsoft's DirectCompute API and AMD's stream processors (FireStream).

Unfortunately, having the hardware in place to execute concurrent operations is not synonymous with improved performance. Applications running on a parallel platform will not benefit directly from the underlying hardware unless explicitly designed to exploit the available functionality. Also, not all sequential algorithms lend themselves to implementation on parallel architectures.

As stated in Chapter 1, this project entails the development of a prototype compiler which allows the programmer to exploit some of the parallel processing capabilities that data-parallel architectures offer. More specifically, the developed compiler targets the CUDA platform.

One of the objectives of this work is to equip programmers with a tool that simplifies the development of algorithms for data-parallel architectures, namely a compiler for a programming language without explicit concurrency constructs, and which takes sequential, functional-style code as input and endeavours to produce fast, parallelised code for CUDA C/C++ as output. In data-parallel problems, much computational power is spent on traversing huge data-sets, and in functional programming this is equivalent to iterating over lists or similar data structures by the means of recursive, referentially transparent functions [Hud89]. Therefore, in order to try to reduce the amount of processing time spent on traversing large data-sets, automatic parallelisation is a matter of converting amenable, recursive functions into parallel loops.

Initially the focus of this work was directed towards the parallelisation phase of the compilation. However, to improve the quality of the automatic parallelisation process, the compiler depends on the outcome of a range of other optimisations. Consequently, the focus has shifted away from the actual parallelisation phase and towards the optimisation techniques that can be employed to prepare the input for parallelisation.

The following sections study CUDA and the complementary CUDA software development kit in greater detail and aim at giving a good foundation for the code generation phase of the compiler. §2.4 and onwards include an evaluation of the optimisation techniques that can be applied to functional-style code to prepare the input for parallelisation. Additionally, influential programming languages and software libraries which are relevant to this work are evaluated and discussed.

2.2 The Compute Unified Device Architecture

CUDA was first released in 2007 and is a data-parallel hardware architecture for graphics processors. In recent years, it has become increasingly popular to utilise CUDA's capabilities in high-performance, general-purpose computing [SK10, §1.3]. The appurtenant CUDA framework consists of a compiler, profiling tools, various software libraries and documentation to help the programmer exploit the full capabilities of CUDA. The compiler is an extended variant of C/C++ which allows the programmer to write heterogeneous code, targeting both CPUs and GPUs. The language adheres to the C/C++ standards when targeting the CPU, commonly denoted the host. However, due to limitations in the GPU hardware, there are some restrictions to what can be achieved in device code. In later releases of CUDA, these limitations have diminished, moving the CUDA language towards full C/C++ support [NVI10b].

In the current version of CUDA, devices with compute capability (a version number identifying the feature set of a particular unit) greater than or equal to 2.0 can employ recursion and indirect function referencing. These two features are fundamental to functional programming [Hud89], and hence, the implementation of these allows for more functional-style code to run on the GPU than what has previously been possible. It is also worth noting that devices with compute capability 2.0 can execute multiple kernels concurrently [NVI10b]. In further work, this can be exploited to implement a limited form of task-parallelism.

2.2.1 The Threading Model

CUDA-enabled GPUs offer a scalable programming model. Each GPU holds one or more streaming multiprocessors (SMs) and can provide up to a total number of 480 processing cores [NVI10b]. When executing programs on such GPUs, one normally spawns a large number of identical threads. Each of these threads is identified by either a one-, two- or three-dimensional index. This indexing system makes it trivial to invoke computations across elements in a domain such as a vector or a matrix, *i.e.*, the programmer can easily assign the computation of each thread to a specific cell or a specific group of cells in, say, a matrix.

The schedulers of these GPUs organise threads into primitive groups, called warps. The threads in a warp get scheduled to run concurrently on a single multiprocessor and are closely related in terms of what resources they share and have access to. However, the programmer never deals with warps directly. The programmer deals with threads and blocks of threads. The number of threads that gets allotted to each block is constant across all blocks and is typically a multiple of the warp size. A choice of too many threads per block, exhausts the registers and the shared memory. Similarly, a choice of too few threads per block yields poor utilisation, *e.g.*, opting for 16 threads will make half of the multiprocessor lie idle since the hardware is designed to run 32 concurrent threads per multiprocessor [NVI10b].

Computational blocks are distributed across grids and are indexed in a similar fashion to that of threads. The maximum dimensions of the grids and blocks are dependent on the underlying hardware and can be retrieved by querying the device. The graphics processor can execute equally many blocks in unison as there are multiprocessors on the chip. Hence, each block gets allotted to a unique multiprocessor. If there are more blocks than there are multiprocessors, the GPU will assign multiple blocks to each of the multiprocessors for alternate execution.

On a lower level, warps of 32 threads each get scheduled for concurrent execution and will run until, say, a lengthy memory access occurs. A warp hitting this condition gets descheduled, allowing for other warps to run until the memory access has been completed. The programmer is generally advised to assign multiple blocks to each multiprocessor since this provides more choices of warps to schedule if a multiprocessor has to wait for a synchronisation to finish [NVI10b]. To reduce the need for synchronisation and idling in general, it should also be noted that the control flow of each thread in a warp should follow the same execution path [NVI10b].

All the threads that get spawned on a CUDA-enabled graphics processor execute the same sequence of instructions, namely a kernel. In CUDA C/C++, kernels are disguised as annotated functions. In general, the programmer decorates the functions with function-type qualifiers to specify where the code blocks reside, *i.e.*, whether they reside on the host or on a GPU device, and where the function is callable from.

On CUDA-enabled GPUs, the spawning of threads is considered computationally cheap. However, it does make sense to reduce the number of control transfers across device boundaries, *i.e.*, between the CPU and the GPU. Multiple code blocks can be grouped together, lessening the scheduling and communication overhead. CUDA implements a barrier function to coordinate communication between threads of the same block [NVI10b]. This can be used to synchronise threads and, hence, resolve data interdependence between operations.

For devices of compute capability 1.3 and higher, double-precision floating-point arithmetic is supported. However, there is a penalty for using 64-bit floating-point operations in terms of clock cycles. The throughput for 32-bit floating-point instructions is the same as the throughput for integer instructions, so integers and single-precision floating-point numbers can be used side by side without affecting the performance. On the other hand, integers and reals should not be used interchangeably as this introduces implicit type conversions to the program, which in turn slows down the execution [NVI10b].

2.2.2 The Memory Model

CUDA threads normally operate on data residing in the physical memory of the graphics processor on which they get executed. That is to say that, unless page-locked host memory is used, the data in the CPU is inaccessible to the GPU, and vice versa.

CUDA implements a shared memory model with several layers of separation. On the highest level, there is a chunk of global memory which is accessible from all the threads. This type of memory is considered slow and expensive due to the latency and the limited bandwidth of the memory buses between the global memory bank and each individual multiprocessor. To maximise memory throughput when using global memory, it is important to follow the most optimal access patterns for the architecture to ensure that the reads and writes can be grouped together and optimised by the GPU. Lower-level banks of cached, read-only memory, such as constant memory and texture memory, can significantly improve the performance of a program [NVI10b].

The host memory can only be accessed directly from a CUDA device if the memory page or the memory pages that contain the data of interest get locked. The locking of host memory can potentially speed up the application. However, by locking the memory, the pages in question cannot be swapped to disk. The physical memory is a scarce resource, and since page-locking reduces the amount of physical memory that is available to the operating system, it can have a severe impact on the overall system performance [SK10].

Location / Scope	Size	Hit Latency	Description
Global (Read/Write)			
Off-Chip / Global	100s of MBs	100s of cycles	Persistent across kernel invocations. More efficient to access when multiple threads access contiguous elements simultaneously – this enables hardware to coalesce memory accesses to the same page.
Local (Read/Write)			
Off-Chip / Function	$\leq$ Global[1]	$=$ Global	Memory used on a per-thread basis. Register usage is to be preferred, but this space can be used for register spilling, *etc.*
Shared (Read/Write)			
On-Chip / Function	16 KB[2]/SM	$\approx$ Register latency	Local space of cached memory that can be shared between threads in a thread block.
Constant (Read-Only)			
On-Chip / Global	64 KB total	$\approx$ Register latency	Data originally residing in global memory, but cached in each SM. Commonly used in lookup tables. Simultaneous requests within a multiprocessor must be to the same location, otherwise delays will occur.
Texture (Read-Only)			
On-Chip / Global	$\leq$ Global	$>$ 100 cycles	Locally cached global memory which capitalises on two-dimensional locality. Hardware interpolation and configurable behaviour of texture edge values make this kind of memory useful in applications such as video encoders.

Table 2.1: Memory properties of typical CUDA-enabled graphics processors (based on Table 1 in [RRB+08]).

Each multiprocessor sits on an isolated chunk of memory, known as the shared memory of that particular unit. Due to the locality and the caching of shared memory, it is much faster to access and use than, for instance, the global memory. However, data stored in this memory bank cannot be shared between separate multiprocessors and has a limited storage capacity [NVI10b].

On a thread level, CUDA offers a set of fast registers and a relatively slow, local memory bank. Registers, which reside physically on each of the processing units of the multiprocessors, are the preferred placeholders for data during computation. It is important to try to keep the number of temporaries to a bare minimum – *cf.* §2.4 on page 16. If the computation requires more temporaries than there are registers available, slow local memory or similar memory banks will have to be used instead. This phenomenon is called register spilling [ALSU06].

In summary, the CUDA architecture offers a lot in terms of processing power. However, the CUDA platform is, as most other parallel architectures, extremely sensitive to bad memory accesses. Data copies are generally more time consuming than computations due to the latency and the bandwidth of the memory buses. Thus, the amount of data to be transferred between the host and the device, and the extent of each of these transfers, should be minimised. Care must be taken when deciding where the data is to be stored. Bad memory access patterns will have severe implications on the performance of the application [RRB+08]. As is the case on most parallel architectures, profiling and experimentation seem to be the best ways of finding the fastest implementation for a particular problem. However, the guidelines and the optimisation strategies provided by NVIDIA [NVI10b, §5] generally yield good results [NVI10b, RRB+08, Har08].

[1] The amount of local memory available to each thread varies between 16 KB and 512 KB. This depends on the compute capability of the device.

[2] Devices of compute capability 2.0 has 48 KB of shared memory per streaming multiprocessor (SM).

2.2.3 NVIDIA CUDA C/C++

CUDA C/C++ implements a conventional C/C++ language with a number of extensions, allowing the programmer to write heterogeneous code that targets both the CPU and CUDA-enabled graphics processors. The language is fully compliant with the C/C++ standards when writing host code, but there are some restrictions to what can be done in device code. Even so, the language is moving towards full compliance for device code as well, but only when targeting devices of compute capability 2.0.

The CUDA language includes type qualifiers to specify where code and data should reside. These type qualifiers can also be used to specify whether to use shared memory, texture memory or constant memory for a piece of data. Additionally, the language provides dedicated constructs for spawning CUDA kernels from the CPU [NVI10b].

As mentioned in §2.2.2 on page 8, all the data that is accessed by a CUDA kernel should reside in the memory of the executing device. Hence, to be able to do any form of data processing, the input data must be transferred from the CPU to the GPU. Similarly, the result must be transferred back to the CPU after the computation has finished.

In other words, the programmer must be able to allocate, read, write and free device memory. Consequently, the programmer must also be able to query for what devices are available in the system on which the target program is to be run. The software libraries that are provided by NVIDIA have a vast collection of functions to aid the programmer in this process.

2.2.4 Challenges

As mentioned in the previous sections, the GPU hardware architecture imposes certain limitations on the programmers. To gain speedups, the programmer is highly dependent on the problems at hand being naturally data-parallel. If they are, the computations will scale well on the CUDA platform as CUDA-enabled GPUs provide a large number of hardware managed threads, each which can compute a certain portion of each problem independently. However, the GPUs are extremely sensitive to irregular memory accesses. Poor utilisation of the caching capabilities of the graphics processors can significantly influence the performance of a program [RRB+08]. Hence, it is crucial to have a good data layout to allow large chunks of subsequently accessed data to be cached on the GPU. The speed and bandwidth of the memory bus between the CPU and the GPU and the memory buses between the global memory on the GPU and each of the multiprocessors, yield a relatively high latency for accessing host memory and global device memory. As discussed in §2.2.2, local, cached memory and on-chip registers have a much lower latency and are to be preferred when choosing where data is to be stored [Har08, RRB+08].

In general, computations are cheap and memory accesses are expensive on these architectures. Sometimes, for smaller problems, one might not even benefit from parallelising a program due to the overhead that is caused by data transfers between the host and the device. It is also worth mentioning that, in some cases, recomputing data can be cheaper than fetching it from memory.

2.3 Languages and Libraries for Parallel Computing

In §2.2 on page 7, we gave an introduction to CUDA, the hardware platform which the compiler – from now on called Fcc – is targeting. The remaining sections of this

review focus on the code transformations that the compiler can perform to parallelise and optimise functional-style code.

Semantic analysis of parallel programs is an area which has been researched for many years, and there have been made numerous attempts to facilitate parallelism in programming languages, CUDA being one of them. In some languages, native constructs have been added to the language to allow the programmer to expose parallelism. More commonly though, is the use of external libraries to achieve the same objective.

This section contains an evaluation of a selection of libraries and languages for parallel computing. The purpose of this investigation was to identify key features which should be considered in the design and implementation of the project.

2.3.1 Shared Libraries and APIs

As discussed in §2.3.2 on page 12, there exist many programming languages which cope with concurrency. However, there are also a lot of libraries available to exploit parallelism in conventional languages. The collection of libraries that is described in this section is a small but illustrative subset of these.

Conventional languages often deal with task-parallelism as opposed to data-parallelism. For instance, in Java, the programmer can extend the `java.lang.Thread` class to run code blocks asynchronously. In C, the programmer has access to threading libraries such as PThreads [POS95]. These libraries allow for concurrent execution of certain types of functions and allow the programmer to manage shared resources by the means of mutually exclusive locks, semaphores and other similar mechanisms.

Even though task-parallelism can be used to solve data-parallel problems, the scheduling and thread management in task-parallel libraries tend to be more costly in terms of performance. Data-parallel architectures, such as CUDA, often manage threads or similar entities in hardware, and this allows for fast spawning and simultaneous execution of a vast number of processes, as mentioned in §2.2.1 on page 7.

In Fcc, only data-parallelism is exploited, so task-parallel libraries are of little interest to us. Some devices of compute capability 2.0 and higher can execute multiple kernels concurrently. Similarly, multiple devices and CUDA streams can be utilised to accomplish a primitive form of task-parallelism [NVI10b]. However, this is a subject for further work. The following subsections focus on data-parallelism and study a selection of libraries that have been relevant to this project.

2.3.1.1 CUBLAS and Accelerator.NET

CUBLAS is a GPU-accelerated implementation of the Basic Linear Algebra Subprograms (BLAS) in CUDA C/C++ [NVI10a], and consists of common linear algebra routines which can be used transparently without having to deal with CUDA directly.

Accelerator.NET is a software library which adds support for data-parallelism to the Microsoft .NET framework [TPO06]. The library provides wrapper classes with generic routines to perform operations on single- and multi-dimensional arrays in parallel. How these computations get distributed onto the machine depends on the underlying hardware, *e.g.*, if the program runs on a quadcore machine, the library splits the workload in four and executes each chunk on a separate core. Accelerator.NET can also exploit available graphics processors.

The concept of dynamic distribution of computation introduces the idea of dynamic execution behaviour. This means that the compiler can add code to decide how to evaluate a function at run-time. Say, for instance, that the programmer wants to evaluate

`map(F, List)`. Then, Fcc can produce code so that `map` executes in parallel for large numbers of elements in `List`, and sequentially otherwise. Hence, the compiler should produce two versions of the function as well as extra code to determine which version to use based on a computational cost model such as, for instance, the Parallel Random Access Machine (PRAM) [Ble96, ST98]. The parallel language NESL, which is presented in §2.3.2.5 on page 16, implements such a performance model and thereby gives the programmer a formal way of reasoning about the space and time complexities of a program [BG96].

Reuse of well-written and fast code is a key concept in both CUBLAS and Accelerator.NET. With respect to Fcc, the compiler should recognise common patterns in the code and apply high-performance code templates, where it sees fit, during the code generation stage. For example, recursive functions reducing the elements of a list into a scalar can be substituted with an instance of, say, the high-performance parallel reduction algorithm that was presented in [Har08].

2.3.1.2 OpenMP

OpenMP is an application programming interface (API) that allows the programmer to perform cross-platform, shared-memory multiprocessing. The library supports the C/C++ and Fortran languages, and is available for many hardware architectures. OpenMP controls parallelism by the means of compiler directives, library routines and environment variables. In other words, OpenMP provides an explicit programming model for parallel computing [ARB08].

In OpenMP, the programmer can decorate looping constructs to make them run in parallel. However, since the order in which the parallel loop gets executed is indeterminable, loops should only be parallelised if there exist no loop-carried dependences. Loop-carried dependence is a common issue one has to tackle when parallelising loops and is further discussed in §2.4.6 on page 22.

2.3.2 Programming Languages

There exists a vast number of programming languages out there, and a substantial subset of these targets parallel computing. In this section, a selection of languages are presented to give a flavour of the style in which parallelism and concurrency can be expressed. More specifically, this section discusses programming languages and their mapping onto parallel problems.

In procedural languages, flow control happens primarily through loops, conditionals and function calls, and the order of execution is vital. Many procedural languages, such as for instance C and C++, supports pointers and aliasing. However, memory indirection introduces additional complexity to the data-flow analysis and, hence, renders the languages undesirable when dealing with automatic parallelisation of loops [BGS94].

Data-flow analysis can be used to identify loop-carried dependences and is therefore an important aspect of this project. Fcc opts for a functional-style language, partially because of the lack of memory indirection in this programming paradigm.

It should also be noted that aliasing makes it difficult to define a complete semantics for a language, making it inherently difficult to reason about a program's correctness [BGS94]. The amenability to correctness analysis is one of the strong sides of functional programming [Hud89], and is particularly valuable to this work as it simplifies the application of cost models when performing automatic parallelisation.

2.3.2.1 Automatic Parallelisation of Fortran

Fortran is an example of a procedural programming language. Originally, Fortran disallowed memory indirections and, thus, was considered to be more amenable to data-flow analysis than its sibling languages, such as, C and C++ [BGS94]. However, current versions of Fortran support pointers and other features that are associated with aliasing [ISO97]. As mentioned earlier, unrestricted pointers introduce difficulty in the deduction of which variable a pointer is referring to and, hence, make state changes unpredictable and difficult to track. This reduces the opportunity of optimisation [BGS94].

Nevertheless, Fortran is still highly regarded in the high-performance engineering and scientific computing communities [BGS94] and has been studied in a number of research projects on parallelisation [ABC+88, BKK+89, PGH+89]. The research that has gone into the parallelisation of Fortran has, *inter alia*, lead to the development of High Performance Fortran (HPC). HPC is an extension of the Fortran 90 compiler which exploits data-parallelism by distributing the workload of array computations and certain kinds of loops across multiple processors [For97]. Some of the concepts that were introduced in HPC, were later adopted in Fortran 95.

Co-Array Fortran is another approach to parallelising Fortran. The general idea behind this Fortran extension is to replicate the compiled program and all its associated data, called an image, and execute multiple such replicates asynchronously. The images can then interact with each other through shared data, and synchronise their execution by the use of barriers [NR98]. This approach is quite similar to the approaches that can be found in other data-parallel languages and libraries.

Fortran code can also be parallelised through the use of the OpenMP API [ARB08] or the PGI CUDA Fortran compiler [PG10]. The optimisation and parallelisation techniques that are employed in these variants of Fortran are of particular interest to this work and is discussed further in §2.4 on page 16.

2.3.2.2 Parallelisation of Java for Graphics Processors

In 2010, Peter Calvert investigated how to automatically parallelise Java byte-code on CUDA-enabled devices [Cal10]. His work focuses on the extraction of trivial loops from the byte-code that gets generated by, for instance, a Java compiler, and on the parallelisation of these loops using the CUDA C/C++ compiler. The tool that was produced as part of Calvert's project allows developers who target the Java Virtual Machine (JVM) to automatically or manually (by the use of annotations) benefit from the parallelisation of looping constructs in their programs.

The data-flow and dependence analyses that are carried out by Calvert's compiler tool have many similarities to the analyses that are undertaken in Fcc, and some of the topics that were discussed in his report are applicable to this project as well. However, a substantial part of his work focuses on topics such as reverse engineering of byte-code, loop detection and kernel extraction from low-level code, and all these subjects are irrelevant to this work.

2.3.2.3 Functional Languages

Functional programming treats computation as an evaluation of referentially transparent functions. Functional languages tend to avoid global state and commonly implement single assignments [Hud89]. The latter also implies immutable data. Explicit looping constructs are absent in these kinds of languages, so looping is accomplished through the application of recursive functions.

Recursion in computer science is similar to the concepts of recursion and induction in mathematics. Therefore, given that there is no state and no mutable data, recursion can significantly simplify the correctness analysis of programs since the programmer can reason mathematically about the implementation [Hud89].

To implement a language which is amenable to flow and dependence analyses and which, consequently, is well-disposed to automatic parallelisation, Fcc opts for a functional-style grammar with strict evaluation, implicit typing, immutable data and no state. Implicit typing is realised by the means of type inference [ALSU06].

Functional languages are claimed to be naturally concurrent, and have been extensively discussed in the literature of parallel computing [BG95, HJ85, ST98, Szy91]. Much of the research has gone into task-parallel issues such as the parallelisation of argument evaluation. Since there are no explicit looping constructs in functional languages, the only way to parallelise loops is to consider recursive functions. This involves that the compiler must transform recursive functions into loops [LH88, Har88], which in turn are amenable to parallelisation.

One of the functional languages which has tried to exploit parallelism is MultiLisp. MultiLisp is a dialect of Lisp which introduces the idea of parallel evaluation of function arguments and so-called futures. Futures are programming constructs which allow the evaluation of certain terms to be performed asynchronously whilst subsequent computations, that are not dependent on the result of the evaluation, are carried out. Hence, the execution of the program can continue until the results of the asynchronous evaluations are strictly needed [HJ85]. However, these extensions introduce side-effects by the use of shared memory and therefore renders the dialect non-deterministic. The approach that is taken in MultiLisp is best suited for task-parallel architectures and thus not particularly relevant to this work.

Lisp is not the only functional language that has been subject to research on parallelisation. Considerable efforts have also been made to try to add concurrency constructs to Haskell [CKLP01, CKL$^+$11, Cha10, TLP02]. Data-parallelism has indeed been exploited in derived versions of Haskell, but mainly by supplying the programmer with library functions to perform parallel operations on arrays and structures. This is similar to what is done in the Accelerator.NET library, which we described in §2.3.1.

However, there are other aspects of the Haskell language that are of interest to us in this project. Haskell implements the call-by-need evaluation strategy. In call-by-need evaluation, arguments do not get evaluated before the function call as they do in call-by-value evaluation. On the contrary, the arguments get evaluated when they are first referenced from within the function body. As opposed to what is done in the call-by-name strategy, where the arguments get reevaluated upon every reference, the call-by-need evaluation caches the results of the first evaluation of every argument so that these results can be reused in subsequent accesses [ASSP96].

The call-by-need evaluation strategy introduces the idea of function memoisation [ASSP96]. If we consider the Fibonacci function in (2.1), we see that evaluating, say, $\texttt{fib}(10)$ will yield multiple calls to all $\texttt{fib}(i), i < 10$. Without memoisation, the computer must evaluate `fib` upon every call, which obviously involves a fair amount of redundant computation. By caching the result of a function call for a given input, the reevaluation of the function will be computationally cheaper since the return value can simply be looked up in a hash-table instead of being recomputed – this is only the case if the cost of recomputation is higher than the cost of a hash-table lookup.

$$\texttt{fib}(n) = \begin{cases} n, & \text{if } n = 0 \vee n = 1, \\ \texttt{fib}(n-1) + \texttt{fib}(n-2), & \text{if } n > 1. \end{cases} \tag{2.1}$$

In light of the objectives that were stated in §1.4, and further justified by the results that can be achieved from applying the optimisation techniques that are discussed in §2.4.3 and §2.4.4 on page 20, function memoisation has not been implemented in Fcc.

2.3.2.4 Concurrency-Oriented Languages

Erlang is a concurrency-oriented programming language which implements strict evaluation, single assignments and dynamic typing. The language was originally designed by Ericsson to support the development of distributed, fault-tolerant applications. Erlang implements an explicit concurrency model based on communicating processes and uses message passing instead of shared, global variables to produce lock-less code [AVWW93].

Another language which deploys concurrency by implementing communication channels between sequential processes, is the imperative, procedural language occam. occam is designed for concurrent programming and has explicit declarations of whether statements are to be executed sequentially or in parallel [SGS95].

In this work we opt for implicit parallelism. Hence, no such constructs have been made available in the Fcc language. One can argue that the compiler should avoid hiding too much away from the programmer [SH92]. However, for automatic parallelisation of loops over large data-sets, a lot can be done implicitly to optimise the program. The applicable optimisation techniques are discussed in §2.4 on page 16.

Communicating processes are not explicitly designed for data-parallel problems and are more relevant to task-parallel computation. In Fcc, parallelisation only occurs in computational expensive sections of the code where a single set of instructions can be applied to multiple data entities simultaneously, *e.g.*, in hotspots such as array-processing loops [ALSU06]. Consequently, other forms of parallelism are not considered.

That being said, Erlang and occam approach concurrency transparently, meaning that parallelism is programmatically exploited in the same way regardless of whether the user deals with a cluster system or a single, unicore machine. This is an interesting concept which Fcc has adopted.

Erlang implements several concepts that help to simplify the analysis and improve the readability of code. Fcc is inspired by Erlang and inherits some of its core features:

- Pattern Matching – Erlang allows for the use of pattern matching in function signatures to improve the readability of function definitions. The language also lets the programmer implement multiple bodies for the same function [AVWW93], *e.g.*, to provide one clause for each possible input case, which, for instance, can make it easier to distinguish between the base case and the inductive step of recursive functions. This approach is analogous in functionality to having an `if`-statement inside a single function body, but simplifies the semantic analysis and improves the readability of the program.

$$\texttt{map}(l, f) = \begin{cases} \emptyset, & \text{if } l = \emptyset, \\ f(l_1) :: \texttt{map}([\, l_2, \ldots\,], f), & \text{if } l = [\, l_1, l_2, \ldots\,]. \end{cases} \tag{2.2}$$

  The `map`[1] function that is described in (2.2) has two discernible types of input, namely l of zero and non-zero cardinalities. The function can therefore be implemented in Erlang by using two separate function bodies, one for each input type, as illustrated in Listing 2.1.

[1] Note that the binary operation :: is given by list prefixation, see §A.7.2 on page 88.

Listing 2.1: The `map` function that is defined in this code snippet, applies a function to all the elements of a list (Erlang)

```
map([], F)    → [];
map([H|T], F) → [F(H)] ++ map(T, F).
```

- List Operations – As shown in the example above, lists can easily be split and examined in Erlang. The construct `[H|T]` splits a given list, l, and assigns the first element of the list, l_1, to `H` and the remainder of the list, $[\, l_2, l_3, \ldots\,]$, to `T`. A similar construct is available in the Fcc language.

  List comprehension is a common feature of many functional and array programming languages [Arm07, Hud89, ST98], including Erlang, and is essentially a programming construct for generating lists based on a set of criteria. This is a concept that is based on the mathematical set-builder notation. List comprehensions are realisable by the application of the fundamental `map` and `filter` functions[2] that can be found in most functional languages, and are therefore parallelisable. List comprehensions are subject to further work.

- Atoms and Variables – Erlang differentiates between atoms, identifiers starting with a lower case letter, and variables, identifiers starting with a capital letter. In Fcc, atoms are automatically substituted with numerical constants at compile-time so that, whenever atoms are used in the program, the machine can compare integers rather than strings. This feature can be regarded as syntactic sugar and is implemented solely to improve the readability of input programs.

2.3.2.5 Performance Models for Data-Parallel Languages

NESL is a parallel programming language which integrates ideas from function programming, parallel algorithms and array programming [Ble95, BHC+93]. The language can be used to solve regular, data-parallel problems, but is also suitable for irregular algorithms that work on sparse matrices and traverse over trees and graphs. NESL follows a divide-and-conquer approach and supports nested parallelism, *i.e.*, the programmer can spawn parallel processes from within other parallel processes [Ble95]. CUDA cannot instantiate new kernels from within executing kernels [NVI10b]. Hence, nested parallelism is disregarded in this work.

As mentioned in §2.3.1 on page 11, an important aspect of NESL is its implementation of a language-based performance model which allows the programmer to perform a formal estimation of the work and depth of a program [BHC+93]. The measures that are yielded by such an analysis correlate to the actual running time on a parallel machine. Therefore, the measures are also useful in the reasoning about a program's performance and, consequently, in the justification of applied optimisations.

2.4 Compiler Transformation Techniques

In this project, we study how functional code can be parallelised. As mentioned in previous sections, loops tend to become the hotspots, *i.e.*, the most computational expensive sections, of the code. To attack this problem when parallelising and optimising functional code, one mainly focuses on the transformation of recursive functions into loops

[2] For instance, the list comprehension $\{\, x^2 \mid x \in L,\ x \leq P \,\}$ can be expressed in any functional language as: `map`$(\lambda x.\, x^2,$ `filter`$(\lambda x.\, x \leq P, L))$.

and then on the parallelisation of these loops, which allows us to distribute the workload across multiple processing units [ALSU06, Har88]. The reader should note that there exist other approaches to parallelisation of functional code, see §2.3.2 on page 12. However, these are predominately related to task-parallel architectures.

Tail-recursive functions are easily transformable into loops [BGS94, Har88]. Augmenting recursion can be converted into tail recursion [LH88, Bir77, LS00], and consequently, by eliminating the resulting tail call, a loop can be derived. However, some recursive functions are not amenable to these optimisations, *e.g.*, consider a function performing an in-order traversal over a binary tree. Fcc addresses the problem of optimising instances of tail recursion and augmenting recursion. Other kinds of recursion, such as tree recursion, indirect recursion and mutual recursion, are not considered. However, the translation of mutual and simultaneous recursion into iteration is a routine extension of the methods that are implemented in Fcc, and should therefore be considered in further work.

One of the first stages of the compilation process is the translation of the input source code into an abstract syntax tree and, subsequently, into an intermediate code representation, *e.g.*, in the form of three-address code [ALSU06]. Three-address code and other intermediate representations are discussed in §2.4.1.

Once the abstract syntax tree has been generated and an intermediate representation has been derived, the compiler can perform certain transformations on these to optimise the output which gets produced in later phases of the compilation [ALSU06]. The following sections study various optimisation techniques that can be applied to functional code, some of which operate on the abstract syntax tree and some of which operate on the intermediate code representation.

Before proceeding, it is worth mentioning that the compiler should always prioritise correctness over performance. The result of a compiler transformation, P', must always be correct in the sense that the program prior to the transformation, P, and P' both produce the same output for the same input, *i.e.*, for all input I, $P(I) = P'(I)$. In the mathematical sense, this is relatively easy to verify. However, in computing, we also have to consider integer overflows, rounding errors and precision errors. Furthermore, since optimisations and compiler transformations can involve reshuffling of instructions, we have to consider situations where P' raises exceptions at different points in the program than P, or where exceptions get raised due to unhandled cases.

2.4.1 Intermediate Representations

In the context of compiler construction, there is a selection of intermediate code representations to choose from. Functional-language compilers often choose to employ continuation-passing style or administrative normal form [App98b, Ken07], whilst imperative-language compilers normally opt for three-address code or static single assignment form. This section presents the intermediate representations that are listed above. The intermediate representation that is used in Fcc is presented in §5.3 on page 40, together with a justification for the choice of representation.

- λ-Calculus – Representations that are based on λ-calculus can be used to represent the input code in functional-language compilers. However, these are not good intermediate representations for call-by-value languages, such as Fcc, as they have the potential to change infinite loops into terminable programs. Furthermore, single evaluation of parameters can be expanded into multiple evaluations [App92].

- Continuation-Passing Style (CPS) – The continuation-passing style employs single assignments and is mainly concerned with interprocedural optimisations. This representation essentially turns all function applications inside out, and has the property of making control-flow and data-flow explicit [App92]. CPS also has the property of ensuring soundness of β-reductions (function inlining). In contrast, call-by-value languages that use an intermediate representation that is based on λ-calculus, only allow for sound applications of the weaker β-value rule since side-effects and non-termination can invalidate the full β-reduction [Ken07].

  CPS represents all arithmetic and logical operations as functions. Thus, all operations appear in the form of function calls. Additionally, all functions take an extra argument, namely a continuation. The continuation of a function is another function whose main objective is to receive and process the return value of the function which is referencing the continuation. Instead of using the conventional call-and-return paradigm, CPS uses continuations to control the flow of the program. Consequently, control will never return to the call-site of a function as the corresponding return value gets propagated through the referenced continuations instead of being passed back to the caller.

  The following example illustrates how an ordinary expression can be translated into continuation-passing style:

$$\begin{aligned} x + (y-1) \times z \quad &\rightarrow \quad \text{add}(x, \text{mul}(\text{add}(y, -1), z)) \\ &\rightarrow \quad \text{add}'(y, -1, \lambda r_0.\ \text{mul}'(r_0, z, \lambda r_1.\ \text{add}'(x, r_1, k))), \\ &\qquad\quad \text{where } \text{op}'(a, b, c) \stackrel{\text{def}}{=} c(\text{op}(a, b)), \text{op} \in \{\text{ add, mul }\}. \end{aligned}$$

  Note that k is the continuation that gets passed in to the evaluation of the entire expression, and that this is the function that is responsible for processing the result of the evaluation of $x + (y-1) \times z$.

- Administrative Normal Form (ANF) – The administrative normal form bears many similarities to the continuation-passing style. For instance, as is naturally the case in CPS, the ANF representation requires all function arguments to be represented trivially, meaning that every argument must either be an immediate value or a variable reference [FSDF93]. This can be accomplished by using the pre-defined `let`-construct, as illustrated below:

$$\begin{aligned} x + (y-1) \times z \quad \rightarrow \quad &\texttt{let } r_0 = y - 1 \texttt{ in} \\ &\texttt{let } r_1 = r_0 \times z \texttt{ in } x + r_1. \end{aligned}$$

  In addition to being extensively used in functional-language compilers, the administrative normal form is also used in the study of semantics for impure functional languages [Ken07, Pit05].

- Three-Address Code (TAC) – Three-address code is a sequence of quadruples holding an operator ($\bullet$), two operands (o_1, o_2) and a destination (d), and is commonly denoted by a sequence of instructions of the form $d \leftarrow o_1 \bullet o_2$. Since each three-address code instruction is limited to having two operands, long and complex expressions must be rewritten in terms of compiler-generated temporaries [ALSU06]. For example, an assignment, $x \leftarrow a + (b \times c) - d$, gets translated into an equivalent sequence of instructions:

$$t_1 \leftarrow b \times c;\ t_2 \leftarrow a + t_1;\ x \leftarrow t_2 - d, \quad \text{where } t_i \text{ are temporaries.} \tag{2.3}$$

Each three-address code instruction can also be assigned one or more labels since the handling of flow-control instructions requires each individual instruction to be addressable.

- Static Single Assignment Form (SSA) – The static single assignment form is a refinement of three-address code, and is mainly concerned with the representation of function bodies. As the name suggests, SSA employs single assignments, *i.e.*, it disallows destructive assignments to variables, and it is therefore well-suited for intra-procedural optimisations and settings where data-flow analyses should be conducted [App92].

  SSA is predominantly employed in imperative-language compilers. Nevertheless, it shares many of the same properties as continuation-passing style and administrative normal form [App98a].

In general, it should be noted that the choice of representation does not have any undesirable effects on the outcome of the optimisation process, as we can easily convert between the different forms during compilation [App98b, App98a, Ken07]. As has been stated in the literature, the functional-style representations, ANF and CPS, and the imperative-style representations, SSA and TAC[3], are essentially doing the same thing, just in different notations [App98b, Ken07].

The main difference between CPS and the other representations that are mentioned above is the need for renormalisation when inlining functions [Ken07]. In CPS we employ continuations, in the form of first-class functions, which receive the results of the corresponding evaluations. However, in the other representations, we introduce new variables. To perform inlining in the latter case, we need to rename all the variables of the function body, *i.e.*, renormalise the function. This contrasts to CPS, where we can perform direct inlining without renormalisation.

2.4.2 Commonly Applied Transformation

This section lists a set of commonly applied TAC transformations. These semantics-preserving transformations are examples of machine-independent optimisations that are applied iteratively until no more optimisation candidates can be found [ALSU06]. The listed transformations assume that the input is in the form of a basic block, *i.e.*, a sequence of TAC instructions without branch instructions, except from potentially at the end of the block. Furthermore, only the first instruction in the block can be labelled.

- Common Subexpression Elimination – In common subexpression elimination, we consider groups of quadruples which all contain the same two operands and operation type. All pairs (q_1, q_2) in such a group get processed. If there are no intermediate assignments to any of the operands of q_1 and q_2 in the instructions linking the two TAC instructions together, the first TAC instruction, with respect to a chronological ordering, is retained, and the second instruction gets substituted by an assignment, $d_{q_2} \leftarrow d_{q_1}$. For example, considering a sequence of TAC instructions, $t_1 \leftarrow a \times b;\ t_2 \leftarrow a \times b$, common subexpression elimination transforms the latter instruction into $t_2 \leftarrow t_1$. Note that, since the Fcc language implements single assignments, no intermediate assignments can occur. Hence, the test for destructive assignments lapses.

[3] Given that the employed TAC representation can guarantee that all assignments are non-destructive.

- Copy Propagation – This transformation exploits the substitutions that are performed by common subexpression elimination. The copy propagation transformation considers all copy assignments, *i.e.*, TAC instructions on the form $d \leftarrow o_1$, and replaces every future reference to d by o_1. *E.g.*, considering the TAC sequence $t_1 \leftarrow a + b;\ t_2 \leftarrow t_1;\ c \leftarrow t_1 \times t_2$, copy propagation transforms the latter instruction into $c \leftarrow t_1 \times t_1$, since $t_2 \leftarrow t_1$ gets propagated through all of the subsequent instructions.

- Dead Variable Elimination – This transformation identifies and eliminates redundant instructions. For instance, in the example above, dead variable elimination will render $t_2 \leftarrow t_1$ redundant since there are no references to t_2 in the ensuing instructions. This instruction will therefore be removed, yielding the following TAC sequence: $t_1 \leftarrow a + b;\ c \leftarrow t_1 \times t_1$.

- Constant Folding – This transformation evaluates and folds all arithmetic operations where both operands are numerical. *E.g.*, $t_1 \leftarrow 5 \times 7$ becomes $t_1 \leftarrow 35$.

- Algebraic Transformation – Algebraic transformation considers and simplifies all redundant arithmetic operations, such as addition by zero and multiplication by one. *E.g.*, $t_1 \leftarrow x + 0$ becomes $t_1 \leftarrow x$, $t_1 \leftarrow 1 \times x$ becomes $t_1 \leftarrow x$, $t_1 \leftarrow 0 \times x$ becomes $t_1 \leftarrow 0$, and so on.

- Strength Reduction – Strength reduction substitutes certain kinds of operations with computationally cheaper operations. For example, addition is a cheaper operation than multiplication. Hence, $t_1 \leftarrow 2 \times x$ can be replaced by $t_1 \leftarrow x + x$. Similarly, $t_1 \leftarrow x^2$ can be replaced by $t_1 \leftarrow x \times x$, and so on.

2.4.3 Tail Recursion Optimisation

As mentioned in earlier sections, tail-call elimination allows us to turn tail-recursive functions into explicit looping constructs [ALSU06, BGS94]. In tail-recursive calls, execution control need not return to the point of invocation. Thus, there is no need for a stack to store return addresses and to pass function arguments. Tail-call elimination converts tail-recursive function calls into normal branch instructions and reassigns the applied function arguments as needed. Similar transformations can be made for head-recursive and middle-recursive functions. However, Fcc only deals with tail recursion.

By the application of procedure inlining, one can turn invocations of tail-recursive functions from within other recursive functions into sub-loops. If this transformation yields a perfect loop nest and the inner loop has a greater iteration span than the outer loop, one can optimise the code even further by performing loop interchanges [BGS94]. A loop nest is a set of loops, one inside another. If the bodies of all these loops, except from the innermost body, only consist of the next loop in the chain, the nest is called a perfect nest. As discussed in §2.4.7.3 on page 25, imperfect loop nests can be turned into perfect nests by loop distribution.

The loop interchange transformation exchanges the indices, the bounds and the step sizes of two nested loops and can be used to rearrange loops so that the outermost loop becomes the most amenable to parallelisation. Since there is no support for nested parallelism in Fcc, such rearrangements can play an important role in the optimisation phase. However, in practise, the Fcc compiler generates few instances of nested loops, and so, the implementation of this feature has not been prioritised.

2.4.4 Optimisation of Augmenting Recursion

Augmenting recursion can be converted into tail recursion by the means of introducing one or more auxiliary parameters to the original function signature [LS00]. This resembles rewriting the functions as loops, similar to those that can be found in imperative languages [Shi05]. The Fcc compiler exploits the trivial cases where such transformations can be made. The chief benefit from converting instances of augmenting recursion into tail recursion is that the resulting instances of tail recursion can be transformed into physical looping constructs. This eliminates the stack growth that is normally associated with recursion [ASSP96, §1.2].

2.4.5 Structural Recursion

On data-parallel architectures, we are mainly concerned with computations that can be executed in parallel over vast data-sets. Hence, an important aspect of this work is to optimise instances of recursion over data structures.

The only data structure that is available in Fcc, is homogeneous lists. Perhaps the most common way to iterate over such lists, is to consider two well-defined entities of the structure, namely the first element (the head) and the remainder of the list (the tail). This allows us to consider two cases of input; the case where the list is empty and the case where the list is built up from an element prepended to an arbitrary list. It is easy to see that this is a recursive structure and, thus, that iteration over such data structures strongly resembles the mathematical approach to structural induction, see the `map` function that was described in (2.2).

The action of prepending updated elements to a recursively processed list, is in fact an instance of augmenting recursion. *E.g.*, in (2.2), the second function clause must process the function call `map`$([\, l_2, \ldots])$ before it can prepend $f(l_1)$ to the resulting list. Consequently, tail-call elimination cannot be applied directly.

A common way around this problem is to perform a transformation called tail recursion modulo cons. The instance of augmenting recursion that is described above depends on the prefixation of an element to the result of a recursive call. Hence, a depth-first traversal is performed and the caller must wait for the callee to finish computing before continuing. However, if we introduce an auxiliary variable and initialise this with the empty list, we can turn the depth-first traversal into an inline, iterative process where the result of the computation that gets carried out on the head element can be appended to the auxiliary list for each iteration [ASSP96, §2.2].

In conventional, functional programming languages, lists are normally represented as recursive structures in the form of linked lists. This representation fits well with the computational model of functional languages. In general, the action of appending elements to a linked list is computationally more expensive than prefixation and, hence, can have a bad impact on our derived optimisation. However, in this context, the resulting loop can be rewritten to allow for efficient suffixation of elements [ASSP96]. Thus, the computational cost of appending elements to the list does not affect the performance of the generated loop.

Generally, linked lists are considered to be computationally efficient data structures. However, in terms of memory usage, they are considered wasteful due to the extensive use of referencing between elements. Each element of the list, l_i, holds a pointer to the rest of the list, $[\, l_{i+1}, \ldots, l_n \,]$. Consequently, the linked list structure has a memory overhead of $O(n)$. Also, the linking of elements is bad with respect to the cost of main-

taining memory coherence, since all the pointers must be updated whenever the list gets moved from one memory location to another.

In CUDA-specific code, we want to reduce the amount of data that gets transferred between the host and the device and, thus, remove aliasing altogether. The use of pointers poses a problem with respect to memory usage and memory coherence. To address this problem, Fcc implements a flat memory model without cross-referencing of elements.

2.4.6 Dependence Analysis

After having converted recursive functions into explicit looping constructs by the means of the methods listed in §2.4.3, §2.4.4 and §2.4.5 on pages 20 to 22, Fcc tries to parallelise these loops. However, to do so, the compiler needs some tools to help determine the parallelisability of the loops. This section introduces one such tool, namely dependence analysis.

Dependence analysis allows the compiler to impose constraints on the execution order of instructions. The result of the analysis, commonly represented in the form of a graph where instructions are denoted as nodes and dependences are denoted as edges, can be used to reorder and, hence, reschedule the execution of instructions to improve the performance of the output program [ALSU06].

2.4.6.1 Control-Dependence and Data-Dependence

There are two types of dependences in computer programs, namely control-dependence and data-dependence. Control-dependence, commonly denoted $S_1 \, \delta^c \, S_2$ for two instructions, S_1 and S_2, deals with the cases where the execution of one instruction *must* precede the execution of another one. For example, the control-dependence relation, $C_1 \, \delta^c \, S_1$, holds for an `if`-statement (C_1) and its true-clause (S_1) since S_1 is strictly dependent on the execution of C_1 [PW86].

Similar dependence relations exist for the operation on data. Data-dependence analysis identifies cases where simultaneous execution of instructions are impossible due to conflicting uses of the same variables, and distinguishes between four different kinds of dependences:

- Flow Dependence – There exists a flow dependence, also called a true dependence, between two subsequent instructions S_1 and S_2 if S_1 writes a value that is read by S_2. A flow dependence $S_1 \, \delta \, S_2$ yields an explicit execution order on the two instructions, namely that the execution of S_1 *must* precede the execution of S_2.

- Antidependence – Antidependences occur whenever an instruction S_2 writes to a variable, v, that is read by a preceding instruction S_1. This is denoted $S_1 \, \overline{\delta} \, S_2$. The restrictions imposed on the execution order by antidependences are not as severe as the ones caused by flow dependences. The antidependence between S_1 and S_2 can be resolved by using two separate memory locations for the variable, v. This allows for S_1 and S_2 to be swapped around without affecting the correctness of the execution [BGS94].

- Output Dependence – If two instructions, S_1 and S_2, write to the same variable, v, there is an output dependence between them, denoted $S_1 \, \delta^\circ \, S_2$. Such dependences can be resolved using storage replication. However, if there are no control

transfers and no intervening use of v between S_1 and S_2, the assignment in S_1 becomes redundant and can be eliminated [BGS94].

- Input Dependence – If two instructions, S_1 and S_2, read from the same memory location, there exists an input dependence between them. This imposes no ordering constraints on the instructions. However, the identification of input dependences can be used to optimise data access patterns and, hence, improve the data locality and cache usage of the program [BGS94].

2.4.6.2 Loop-Carried Dependence

The dependence relations that have been presented so far in this section can be used to determine and optimise the execution order of loop-free sequences of instructions. However, since Fcc is capable of generating iterative constructs from recursive functions, the Fcc optimiser must also encapsulate and consider loop-carried dependences. In loops, dependence relations can bind both instructions and instances of instructions. To separate between the two, the latter is distinguished by a superscript notation stating the loop iteration for which the instance is valid, *e.g.*, $S_a^{(1)}\ \delta\ S_b^{(1)}$ for a loop iterating over values of a variable, i, and in the case where $i = 1$. This notation can also be extended to denote instances of instructions in loop nests [PW86], *e.g.*, $S_a^{(3,5)}\ \delta\ S_b^{(3,5)}$ when $i = 3$, $j = 5$ for an outer loop iterating over i and an inner loop iterating over j.

A further extension of the notation above allows us to easily denote more general dependences between iterations, *i.e.*, loop-carried dependences. If, for instance, the read of an operand in instruction $S_2^{(i)}$ depends on the write of an instruction $S_1^{(i-1)}$, we can denote this by $S_1\ \delta_<\ S_2$ where the less-than sign in the subscript stands for $(i-1) < i$. $\delta_=$ is used to denote dependences within the same iteration. This allows us to annotate dependences in loop nests as well, *e.g.*, $S_1\ \delta_{<,=}\ S_2$, where $<$ is annotating the outermost loop and $=$ is annotating the innermost loop.

Having these tools in place, we can analyse the correctness of permutations in the instruction ordering. This allows for further exploration and evaluation of applicable optimisations [ALSU06]. In Fcc, the dependence analysis is also used to determine whether or not:

- Multiple parallel blocks can be merged together.
- Synchronisation barriers are needed between separate loops within the same kernel due to dependences between instructions in the various loop bodies.
- Allocated memory for data structures, such as lists, can be reused. An example would be the application of a function, f, to all the elements of a list, see (2.2). If the original list never gets used after this operation, the `map` function can simply do inline substitutions into the original data structure instead of creating a new list and populating this with the resulting values. This implies less pressure on resources.

2.4.6.3 Invalidation of Loop-Carried Dependence

Given a `for`-loop of similar form to the one in Listing 2.2, *i.e.*, with two subsequent array accesses and integer constants c, d, j and k, we can readily check whether the loop carries any dependences. Initially, since we prioritise correctness over performance, we take a pessimistic stand and assume that there exists a loop-carried flow dependence in

the loop. However, for the loop to carry any flow dependences, explicitly between S_2 and S_3, the greatest common divisor of c and d, gcd(c, d), must divide $(k - j)$ [AK81, BCKT79, Wol90]. Note that this criterion lapses if the greatest common divisor is zero.

Listing 2.2: The loop only has a loop-carried dependence if gcd(c, d) divides $(k - j)$.

```
S1 : for i in (lbound, lbound+1, ..., ubound):
S2 :   a[c*i+j] = ...  # write to array
S3 :   ... = a[d*i+k]  # read from array
```

For example, if we consider the loop in Listing 2.2 with $c = d = 2, j = 0$ and $k = 1$, we observe that there will be no loop-carried flow dependences between S_2 and S_3 since gcd$(2, 2)$ fails to divide $(1 - 0)$. In other words, the initial assumption of there being a loop-carried dependence between S_2 and S_3 has been invalidated. Thus, the loop is fully parallelisable and can benefit from running on a CUDA-enabled device, given that the work range of the loop is greater than some predefined parallelisation threshold.

2.4.7 Loop Transformations

Up to this point, the compiler has received directions on how it can produce an optimised sequence of instructions from functional-style code. Furthermore, it has received input on how it can convert recursive functions into loops. However, to produce good data-parallel representations of these loops, it might be necessary to apply certain loop transformations [BGS94, BCKT79, Har88, PW86]. Again, it is worth emphasising the importance of loop optimisation and parallelisation, as computer programs spend the majority of their execution time on computing loops. The following subsections discuss the loop transformations that are relevant to this work in greater detail. Most of the loop transformations that have been omitted from this review, are left out due to the fact that the recursion-to-loop conversion renders them inapplicable or effectless.

2.4.7.1 Map and Reduce in Functional Programming

As has been discussed in previous sections, the compiler has to deal with cases where there exist loop-carried dependences. With regards to loop parallelisation, it is worth noting that the workload can still be split up between threads even if there exist interdependences between the iterations, *e.g.*, consider parallel reduction and parallel scan. Borrowing ideas from the functional paradigm, modified versions of the `map` and `reduce` functions can be used to distribute the workload across multiple processing units [DG08, YTT$^+$08]. However, in parallelising the reduction code, it is important to ensure that good coding practises are applied to avoid irregular memory accesses [Har08].

2.4.7.2 Loop-Invariant Code Motions

An expression which appears inside a loop body, but whose value remains fixed for all values of the induction variable, is called loop-invariant. Since loop-invariants get reevaluated for every iteration of the loop, they constitute an unnecessary overhead in execution time. This overhead can be eliminated by hoisting the loop-invariants out of the loop. Such transformations are called loop-invariant code motions [BGS94].

Loop-invariant conditionals which reside inside loop bodies can also be hoisted out of the loop. This involves a bit of extra book-keeping as the encapsulating loop must

be substituted by a conditional and two identical loops, one for the true-branch and one for the false-branch of the conditional (non-existent false-branches are obviously omitted). The process of lifting loop-invariant conditionals out of the loop is called loop unswitching.

2.4.7.3 Merging and Splitting of Loops

Loop distribution, also called loop splitting and loop fission, splits a loop into multiple, smaller loops, *e.g.*, one per instruction found in the loop body. This transformation is suitable for vector machines where parallelisation is exploited on an instruction level. However, on high-level architectures this would increase the loop overhead.

On platforms such as CUDA, it is more sensible to try to increase the number of computations that get carried out per loop iteration, *i.e.*, the number of instructions per kernel [NVI10b]. This can be achieved by loop fusion. Loop fusion merges loop bodies together and coalesces the loop bounds. This reduces the loop overhead and improves the load balance of parallel loops [BGS94]. Furthermore, loop fusion reduces the amount of branching and, hence, improves the instruction pipelining on the machine. With respect to memory, this transformation tend to improve register and cache locality [ALSU06].

2.4.7.4 Unrolling and Rerolling of Loops

The idea behind loop unrolling is to replicate the loop body to reduce the number of iterations and to increase the amount of processing per step. This decreases the loop overhead that is incurred by branching and evaluation of conditionals [BGS94], and improves register and cache locality [ALSU06].

In some cases, the programmer has manually unrolled a loop by hand. Certain compilers revert these cases by rerolling the loops before carrying out any further compiler transformations to the program. In doing so, the compiler reduces the complexity of the input and, therefore, simplifies the subsequent optimisation phases [BGS94].

CUDA C/C++ implements automatic loop unrolling for small loops with known trip counts and provides a compiler directive (`#pragma unroll`) to control the unrolling behaviour of larger loops [NVI10b]. Hence, loop unrolling is not implemented in the Fcc compiler.

2.4.7.5 Loop Coalescing and Loop Collapsing

When traversing multi-dimensional data, we often find ourselves constructing perfect loop nests containing one loop for every dimension of the considered data-set. In these cases, given that the treated data structures allow us to, we can benefit from merging the loops and converting multi-dimensional indices into linear indices [BGS94, ALSU06]. Successful application of this transformation yields less condition checking and branching and can therefore improve the overall performance of the program. This technique is not directly applicable to Fcc since the language only supports one-dimensional arrays. However, the transformation is effectual and should be considered in further work.

2.4.8 Procedure Call Transformations

There exists a number of different interprocedural optimisation techniques, *e.g.*, procedure inlining (§2.4.3 on page 20), procedure cloning (§2.3.1.1 on page 11) and function memoisation (§2.3.2.3 on page 14). Procedure inlining and procedure cloning have, as

illustrated in previous sections, applications that can affect and improve the result of other optimisations, and have been implemented in Fcc. However, other interprocedural optimisation techniques have not be considered in this work.

2.5 Performance Cost Analysis

Loops should only be parallelised if the parallelised version runs faster than the original, sequential version. Therefore, to be able to decide which version to use, we need some way of telling which one is faster. In §2.3.2.5 on page 16, we studied NESL and discovered its implementation of a language-based performance model [BG96, BHC+93]. Such performance models allow us to reason about the performance of sequential and parallel code and can therefore aid us in the process of deciding whether or not to run a loop in parallel. Fcc implements a simplistic performance model which is inspired by the one that is implemented in NESL.

Another point worth noting, as we touched upon in §2.3.1 on page 11, is that the parallelisability of a code section might be dependent on the range of a loop or the input parameters of a function. Therefore, the execution behaviour should be determined at run-time. To address this, Fcc generates a run-time check which determines whether or not to parallelise a loop based on the size of the input and on the computational cost of the loop body.

2.6 Data Layout and Memory Access Patterns

Previous sections have discussed the importance of a good memory layout and well-behaved memory access patterns. The key decision one has to make in attacking this problem, is how to decompose the data when distributing the workload across multiple CUDA threads [NVI10b, Har05, Har08].

In this work, we only deal with data along one dimension since the type system of Fcc is restricted to dealing with integers, floating-point numbers and flat lists (in the form of fixed-size arrays). Hence, the data is decomposed along a single dimension, and consequently, only the outermost loop in a loop nest is considered for parallelisation. In other words, there is no need for block decomposition of data in Fcc, and we only need to worry about data locality along one axis.

It is important to consider the scenario where the number of iterations of a loop exceeds the maximum number of resident threads on the GPU. If the number of iterations is less than the capacity of the GPU, we opt for a serial decomposition of data. Otherwise, a cyclic decomposition is chosen [BGS94]. If the latter is the case, each CUDA thread gets allotted several iterations of the original loop to process.

In §2.2.2 on page 8, we mentioned the importance of data locality and good utilisation of memory caches in CUDA. Deducing whether read-only memory can be used and where the data should be stored to get the most out of the caching mechanisms of the GPU, is crucial as this can significantly affect the performance of the program [RRB+08, SK10].

2.7 Summary

This project involves the development of a compiler for functional-style code whose main aim is to simplify the development of algorithms for data-parallel architectures.

The compiler targets the CUDA platform and endeavors to produce fast, parallel code which exploits CUDA's high-performance computing capabilities.

The first sections of this literature review looked at the CUDA platform in detail, focusing on its threading model and its memory architecture. In short, we observed that CUDA is a scalable and massively data-parallel architecture which is, as most other data-parallel architectures, sensitive to irregular memory accesses and poor utilisation of the built-in caching mechanisms.

The ensuing sections explored the world of parallel software libraries and programming languages to gather insight into how these map onto data-parallel problems. We looked at NESL and how it implements a language-based performance model to reason about the time and space complexities of parallel implementations. Furthermore, we looked at the evolution of parallelisation in the Fortran programming language, which is predominantly concerned with static code analysis. Additionally, ideas that are concerned with the readability of code and the amenability to semantic analysis were collected from various concurrency-oriented and functional languages.

In §2.4, we justified how automatic parallelisation of functional-style code is a matter of converting recursive functions into parallelisable loops and, furthermore, discussed different techniques that can be applied to do so. This section also detailed the set of compiler transformations and optimisation techniques that is implemented in Fcc. Data-dependence and data-flow analyses were discussed in detail, and we saw examples of scenarios where such analyses are useful to the compiler.

The review then moved on to outlining the application of performance cost models, before concluding with a recap on the importance of a good data layout and well-behaved memory access patterns.

We have seen that automatic parallelisation of code is a nontrivial task, and that it is unfeasible to find an optimal solution as such. Again, we emphasise the importance of correctness and repeat that correctness must be prioritised over performance.

CHAPTER 3

Requirements Specification

Contents

3.1 General Overview

This section lists the overall requirements for the compiler system. The rest of the chapter presents the identified software and hardware requirements.

- Correctness – All generated programs must be correct in the sense that they always compute the correct answer, regardless of whether or not they have been subject to optimisation and parallelisation.
- Performance – If parallelism can be beneficially exploited, the generated programs must achieve an improvement in execution time. That being said, a slight decrease in performance should be allowed in boundary cases, considering that the runtime checks for whether or not a loop is parallelisable with the given input take non-zero time.
- Verifiability – The achieved results must be verifiable through testing. Hence, a code base of example programs must be made available to support objective testing.
- Feedback – The compiler must provide sufficient feedback to the user on the results of the type deduction phase and the optimisation phases. Furthermore, error and warning messages must be printed if the provided input program does not comply with the language specification (Appendix A) or if the program contains unused variables or functions, or unreachable code.

3.2 Functional Requirements

The Fcc compiler targets CUDA-enabled devices, more specifically devices of compute capability greater than or equal to 1.3. However, a substantial amount of the work that has been carried out in the course of this project has gone into studying the optimisations that can be performed to prepare the input for parallelisation, rather than the parallelisation itself. Thus, we find that many of the optimisation techniques that are discussed here are applicable to other data-parallel platforms as well.

In terms of functionality, the bullet points below summarise the features that have been prioritised in the implementation phase of this project. Some of the findings have not made it into the implementation of the prototype compiler. These findings are discussed in Chapter 7.4.

(1) Input and Output – The compiler must accept functional code as input, and produce optimised CUDA C/C++ code as output.

(2) Functional Language – The designed input language must comply with the language specification given in Appendix A. It must disallow destructive assignments and prohibit the use of global state. The language must also implement pattern matching on numerical and vector-based function parameters.

(3) Type System– The language must implement a static and implicit type system with support for the types: integers, floating-point numbers, and immutable, homogeneous lists. Additionally, the language must implement support for higher-order functions. Closures are outside the scope of the project and should not be supported.

(4) Type Inference – The compiler must be able to deduce the type of variables based on variable initialisations, applied arithmetic operations, *etc.* In programs, the typing of the data that is being processed can have a severe impact on the performance, *e.g.*, floating-point operations are generally slower than integer operations, and there are penalties associated with implicit type conversions at runtime. Consequently, the compiler should opt for cheaper types and minimise the ramifications of type conversion in the generated output programs.

(5) Optimisations – The compiler must apply the elementary compiler transformations that are described in §2.4.2 on page 19. Additionally, the compiler should implement procedure inlining and loop fusion.

(6) Recursion – The compiler must be able to automatically generate parallel looping constructs from tail-recursive functions. The compiler should also be able to convert well-defined instances of augmenting recursion into tail-recursive equivalents and, consequently, into loops.

(7) Mapping and Reduction – The compiler must be able to identify instances of structural recursion and transform direct, one-to-one maps into parallel mapping constructs. The compiler must also be able to transform additive and multiplicative reduction operations into their parallel equivalents.

(8) Portability – The compiler must generate code for both the CPU and the GPU so that the program can run regardless of what hardware is installed on the target machine, using the CPU as a fall-back option. The output executable must be able

to determine whether to run the sequential or the parallel version of a function based on the available hardware and on the size of the input data.

The following extensions were originally planned, but have not been implemented due to their lack of practical relevance.

- Memory Access Patterns – The compiler should, to some extent, be able to identify whether to flag data for storage in constant memory, texture memory, device memory or shared host memory – due to the penalties that are associated with hits on non-cached memory, the type of the memory that is used by the programmer often affects the execution speed of the program. The programmer should be allowed to override these memory flags, and cases that are not recognised by the compiler should be handled by the use of annotations.

  Due to the immutability of lists in the designed language, see (3) above, it seems appropriate for the compiler to ignore this requirement and to stick to one storage strategy for all the data that is shared between the host and the device.

- Thread Synchronisation – The compiler must be able to detect whether or not a thread synchronisation point (also known as a barrier) is required between any of the statements in a block of GPU code. The detection must be based on a dependence analysis to see whether or not there exist interdependences between the statements in question. The early versions of the compiler should simply divide such statements into separate CUDA kernels.

  If we consider the compiler transformations that are to be implemented in the Fcc compiler and their effect on the generated CUDA kernels, we find that this feature can be disregarded. However, synchronisation should be implemented in further work as other planned transformations are reliant on its implementation.

Listed below are the limitations of the implemented compiler; meaning functionality that has not been considered as part of the project.

- The only data types available to the programmer are: integers, reals, homogeneous lists and function literals.
- The language implements only a limited subset of the arithmetic operations and boolean predicates that are available in other languages, namely: addition, subtraction, multiplication, division, modulo, equality test and comparison.
- The compiler does not employ high-level programming concepts such as object-orientation, inheritance, templates, libraries, structures, *etc.*

3.3 Non-Functional Requirements

The software and hardware requirements for the compiler system are listed below:

(a) The compiled output must target CUDA-enabled systems of compute capability 1.1, but should focus on exploiting features that are available on systems of compute capability greater than or equal to 1.3.

  Compiled programs must be runnable on the x86 machine architecture. However, they obviously require that a CUDA-enabled graphics processor is available on the run-time system to be able to capitalise on the applied parallelisation.

(b) The compilation process is dependent on having access to the CUDA Toolkit, or more specifically the CUDA C/C++ compiler. Thus, the user must have the CUDA Toolkit installed on his or her machine to be able to run the compiler.

(c) The compiler is implemented in Python 3 and uses the latest Python Lex-Yacc (PLY) module. Thus, Python 3.x and PLY 3.4 or higher must be installed on the user's system.

There are no requirements regarding the running time of the compilation process. Nevertheless, it is worth noting that the compiler should guarantee termination of the type stabilisation phase and that the optimisation phases will be aborted after a pre-defined number of iterations if no satisfactory solutions can be derived.

3.4 Testing and Resources

This section specifies the testing requirements for the implemented solution and lists the hardware resources that should be used in the performance testing.

- Verifiability – As mentioned in §3.1, the implemented solution must be verifiable and, thus, an extensive code base of example programs must be made available.
- Correctness – All compiled programs must be correct – in the sense that has been defined previously. Hence, concise unit tests and code snippets must be provided to allow for objective correctness testing.
- Hardware – With respect to quality and objectivity, the performance tests should be carried out on multiple CUDA-enabled machines. More specifically, the compilation results should be subject to testing on the following devices:
  - CUDA-enabled GPUs of compute capability ≥ 1.1:
    Testing should be carried out on graphics processors implementing the NVIDIA GeForce GT 330M (512MB) chipset.
  - CUDA-enabled GPUs of compute capability ≥ 2.0:
    Testing should be carried out on graphics processors implementing the NVIDIA GeForce GTX 480 (1536MB) chipset.

CHAPTER 4

High-Level Design

Contents

4.1 Overview

This chapter aims at giving a high-level overview of the Fcc compiler system. We start out by describing the tool chain and the modules that have been used by the prototype compiler. Then, we justify the design of the input language and give a brief overview of the lexical, syntactic and semantic analyses, before concluding with a high-level overview of the type inference, optimisation, parallelisation and code generation phases. Chapter 5 describes the compilation pipeline in more detail.

4.2 Tools and Modules

As illustrated in Figure 4.1, the Fcc compiler executes on top of the Python 3 interpreter. The compiler uses the Python Lex-Yacc (PLY) module to perform the lexical and syntactic analyses – also known as the tokenisation and parsing phases of the compilation process, respectively. These phases are described in further detail in §4.3.

After having obtained an abstract representation of the meaning of the input program, the optimiser and the compiler back-end collaboratively do their jobs in producing an equivalent, optimised CUDA C version of the program.

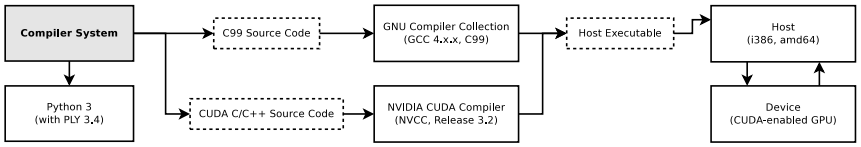


Figure 4.1: This diagram illustrates the compilation process from a tool perspective. The Fcc compiler runs on top of the Python 3 interpreter and outputs C99 and CUDA C/C++ source code. The Fcc ouput is automatically compiled using either the GNU C compiler or the NVIDIA CUDA compiler, culminating in a host executable for x86 machine architectures.

When targeting sequential architectures, the compiler produces C code complying with the C99 standard. The GNU Compiler Collection (GCC) is then used to compile the produced code into a host executable for either the x86-32 and x86-64 machine architecture. Parallelised code is compiled using the NVIDIA CUDA compiler, which essentially is an extension to any underlying C compiler system, *e.g.*, GCC. The output is still targeting the x86 architecture, but the executables are now able to transfer control to all the GPUs that are available on the host system, and to exploit the data-parallel processing capabilities of these devices.

4.3 Language Design

The Fcc language is a functional programming language which implements static and implicit typing. The language implements single assignments and call-by-value evaluation, and is built up from two base types, namely integers and reals. The language can also deal with homogeneous, immutable lists. Furthermore, Fcc supports the use of higher-order functions, meaning that function references can be passed as arguments, stored to variables and returned from functions.

The design of the Fcc language is grounded in the ideas of simplicity and expressiveness. One of the main criteria for the design of the language has been that of maintaining a simple and concise, but yet powerful and expressive grammar. The implementation of pattern matching on function signatures and the allowance for multiple function clauses are examples of this, as they extend the syntax of the language, but with the objective to simplify the coding style. The choice of using call-by-value evaluation is again justified by the focus on simplicity. Lazy evaluation introduces additional complexity to the compiler, the same does support for closures. A formal description of Fcc can be found in Appendix A.

4.4 Software Architecture

The compilation process can be regarded as a pipeline process of well-defined and independent components, as illustrated in Figure 4.2. The pipeline takes functional-style code as input and produces a compiled and optimised executable as output. The first step in this process is the analysis of the meaning of the input program. This is handled by the lexer and the parser.

After the parsing phase is completed, a preliminary semantic analysis is carried out, during which semantic information is added to the derived representation of the input.

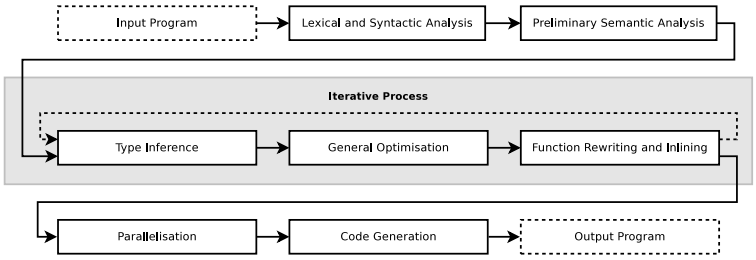


Figure 4.2: This figure provides a high-level overview of the Fcc software architecture. We observe that an iterative process of type deduction and optimisation is executed on the abstract syntax tree that gets generated during the syntactic analysis, and that the result of this iterative process yields an intermediate representation amenable to parallelisation and code generation.

The output of this preliminary analysis is a structural representation of the program, in the form of an abstract syntax tree.

Since the Fcc language is implicitly typed, there is no type information available at this stage of the process. Thus, the compiler passes the syntax tree on to the type inference system, which iteratively tries to deduce the type of all variables and functions in the input program. This stage is generally seen as a part of the semantic analysis.

Once the type information for all symbols is available, we can proceed to the optimisation phase. First, since three-address code is a data structure of lower complexity than syntax trees and, thus, more amenable to analysis and optimisation[1], the optimiser translates the abstract syntax tree into three-address code, as described in §2.4 on page 16. Second, compiler transformations such as common subexpression elimination, copy propagation, dead variable elimination and constant folding are applied. These transformations are applied iteratively until no more optimisation candidates can be found. After the optimiser has finished simplifying the structure of the abstract syntax tree, it tries to derive further optimised versions of the tree by the application of procedure inlining and function rewriting.

Now, if the intermediate representation of the program has changed since entering the type inference and optimisation stages of the compilation process, a new optimisation pass is carried out, meaning that the derived representation gets injected into the pipeline at the point right after the lexical and syntactic analyses have finished.

Once the type deduction and optimisation processes have either stabilised or reached a pre-defined iteration threshold, the optimised, intermediate code is passed on to the paralleliser. This component essentially turns loop candidates into parallel loops whose iterations get distributed across several processing units when executed on the GPU. By loop candidates we mean loops whose bounds, step sizes and loop bodies are well-defined and contain no undesirable interdependences between iterations. In the current implementation of Fcc, the parallelisation phase is mainly concerned with the translation of mapping- and reduction-style problems into their parallel equivalents.

The final step of the pipeline is the code generation phase. The code generator pro-

[1]It should be noted that the abstract syntax tree and three-address code representations that are used in Fcc are equivalent data structures (§5.3.2 on page 40), and that the type inference system therefore also accepts input on the form of three-address code.

duces C99 and CUDA C/C++ compliant source files which are equivalent to the processed input program, but which are based on the optimised intermediate representation. The compiler then invokes the external build tools to produce the desired output program from the generated source files.

4.4.1 Semantic Analysis

The compiler front-end is normally split into two separate components, namely a lexer and a parser. The lexer, also called the tokeniser, is responsible for breaking the input code down into distinguishable entities. These entities are usually called tokens.

Given a set of tokens, the parser tries to deduce the semantic meaning of the program based on a grammar. This grammar is a set of rules that specify how these tokens can be put together. A syntactically correct program is defined to be a sequence of tokens that satisfy a well-defined derivation of these rules [ALSU06].

Strictly speaking, the parser is responsible for the syntactic analysis. To be able to deduce the exact meaning of a program, a more extensive semantic analysis is needed. The semantic analysis is the process of building up a comprehensive symbol table and adding semantic information, such as symbol information and type information, to the syntax tree that gets derived during the syntactic analysis.

4.4.2 Type Inference

As part of the semantic analysis, the types of all functions and variables in the input program are deduced. This process is dealt with by the type inference system. Since Fcc is implicitly typed, the deduction of type information is an iterative process where the compiler tries to assign types to symbols so that the program is in a well-defined state. The aim of this stabilisation process is to establish a set of type assignments that yields a sound interpretation of the language, meaning that the type model in §C.1 on page 97 is satisfied for the derived type assignments.

4.4.3 Optimisation

The optimisation phase is a fine-grained, iterative process consisting of multiple sub-processes. Some of these sub-processes include the commonly applied compiler transformations that are described in §2.4.2 on page 19, the interprocedural optimisations that are mentioned in §2.4.8 on page 25, and the loop transformations that are mentioned in §2.4.7 on page 24.

Given a typed syntax tree, the optimiser attempts to derive a simplified structure by applying the transformations that have been mentioned earlier. If the structure of the intermediate code representation changes as a result of the transformations that get applied in an iteration of the optimisation process, the compiler attempts to run another iteration. The compiler repeats the cycle until no more improvements can be made, or an iteration threshold has been reached.

4.4.4 Parallelisation

After all the type information has been deduced and an optimal intermediate code representation has been derived, the compiler moves on to the parallelisation stage. By this point, amenable recursive functions have been translated into looping constructs. This stage is therefore a matter of analysing the loop bodies to try to find candidates

for parallelisation. More explicitly, this means that an analysis is carried out to identify occurrences of loop-carried dependence. Such dependences put constraints on how the computation of the loops can be distributed across multiple processing units and, thus, determines to what extent the loops can be executed in parallel.

The parallelisation phase is mainly concerned with the search for occurrences of mapping and reduction problems and, thus, focuses on data-parallel operations that can be carried out on list structures. Parallelisable loops are flagged so that the compiler can take appropriate action and produce CUDA kernels for these loops during the code generation phase.

4.4.5 Code Generation

The final step of the pipeline process, before the results get passed on to the external build tools, is to generate code that represents the derived, optimised solution. As mentioned in the beginning of this chapter, Fcc produces C99 and NVIDIA C/C++ code by default.

The code generator is a compound of multiple code emitters, one for each of these supported output formats. The design of the compiler allows for easy construction of new code emitters. Thus, targeting other, currently unsupported architectures, such as OpenCL, requires minimal effort.

The code generation phase is essentially an in-order, depth-first tree traversal where each code emitter outputs appropriate code blocks upon every node visit. The emittance process is for the most part template-based. For more details, refer to §5.12 on page 57.

4.4.6 External Build Tools

Upon the completion of the code generation phase, the resulting code gets passed on to the GCC and CUDA compilers. Fcc ensures that the build tools get invoked with the correct flags and completely hides the use of the external build tools from the end-user.

CHAPTER 5

Detailed Design and Implementation

Contents

5.1 Overview

This chapter details the low-level design and implementation of the prototype compiler that has been discussed in previous chapters.

In short, the compiler takes Fcc source files as input and produces C99 and CUDA C/C++ files as output. In the process of transforming the input into parallel CUDA code, an intermediate code representation of the semantic meaning of the program is needed. The translation of input code into such an intermediate representation is carried out by the compiler front-end.

The intermediate code is subject to various compiler transformations in an attempt to make the code amenable to automatic parallelisation. As mentioned in earlier chapters, Fcc focuses on the parallelisation of data-parallel constructs. The middle-end's main objective is therefore to turn recursive functions into loops and to try to parallelise these. A loop is naturally data-parallel if there exist no data-dependences between iterations. Thus, a data-dependence analysis is required during the optimisation and parallelisation stages to be able to detect parallelisable loops.

A compiler transformation may yield other optimisation candidates, *e.g.*, copy propagation often yields candidates for dead variable elimination. Thus, the optimisation phase is an iterative process that will keep running until a stable, optimal solution has been found or an iteration threshold has been reached. When no more optimisation and parallelisation candidates can be found, the code generator gets invoked and traverses the internal representation of the input program to produce the required C and CUDA C/C++ source files.

5.2 Tokenisation and Parsing

The compiler front-end is divided into two components, namely a lexer and a parser. The lexer translates the input string into tokens, and the parser builds up a syntax tree from these tokens based on the formal grammar of the input language. This section describes these two processes. The data structures and the auxiliary modules that are needed to arrive at an intermediate representation of the input, is discussed in greater detail in §5.3 on page 40.

5.2.1 Lexical Analysis

The lexer translates the string representation of the input code into a list of atomic components, normally called tokens. Examples of such tokens are identifiers, parenthesis, numbers, *etc.* The process of translating the input into tokens is called the lexical analysis of the input. The resulting tokens feed directly into the syntactic analyser, also known as the parser, which studies and analyses the composition of tokens and whether it adheres to a formal grammar or not.

5.2.2 Syntactic Analysis

The parser produces a syntax tree based on the sequence of tokens that gets derived from the input program. This syntax tree is untyped and contains only the most essential in-

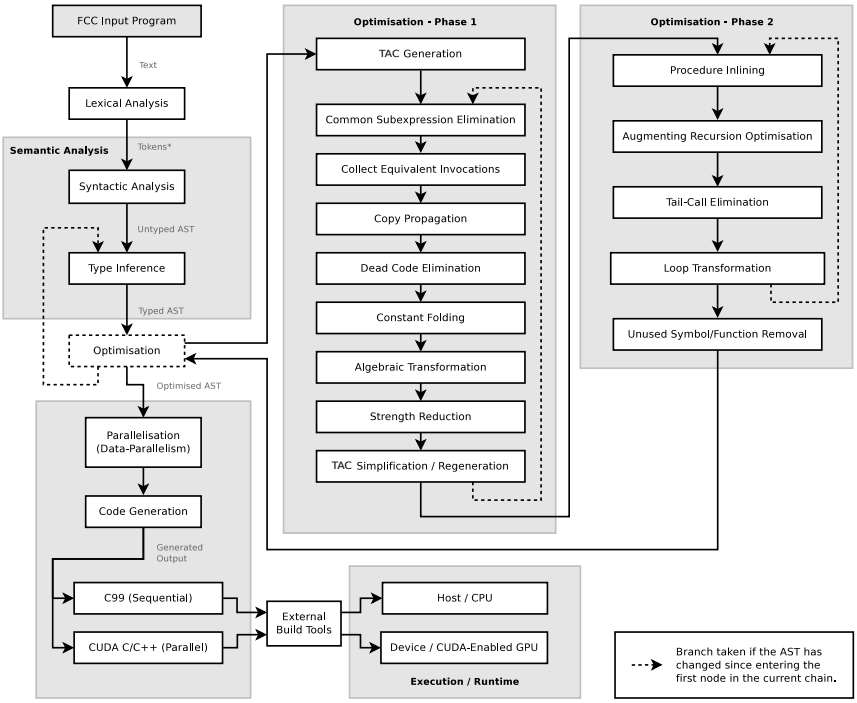


Figure 5.1: The depicted flow diagram provides an architectural overview of the compilation process. The compiler front-end is responsible for the semantic analysis of the input program and, thus, also the type deduction process. As illustrated in the diagram, the optimisation stage is divided into two sub-phases, one dealing with the machine-independent transformations that take place on a block level, *i.e.*, in branch-free sections of the code, and one dealing with interprocedural optimisations and loop transformations. The optimiser and the paralleliser constitute the compiler's middle-end. The back-end is essentially the code generator which produces sequential C99 and CUDA C/C++ code. The produced source files are automatically and transparently compiled using external build tools, yielding the desired executables.

formation about the input program. It should be noted that all identifiers and keywords are stored to a symbol table to ensure unicity of strings and symbols in the internal data structures of the compiler.

The compiler is implemented in Python using the functionality that is provided by the PLY module. The PLY module allows us to implement the lexer and the parser using annotated functions, or more specifically, using Python's support for *documentation strings*. The implementation of the lexical and syntactic analysers can be found in the source files "`parser/lex.py`" and "`parser/yacc.py`."

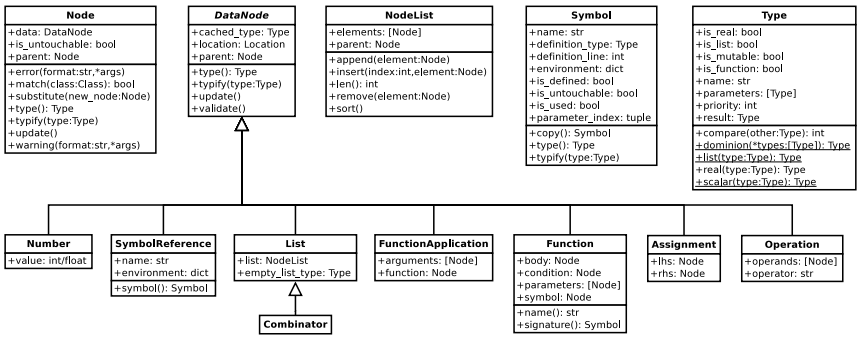


Figure 5.2: This diagram is an excerpt of the UML diagram of the implemented solution, with a particular focus on the data structures that are used in the intermediate code representation.

5.3 Intermediate Representation

The compiler front-end generates an intermediate code representation in the form of an extended abstract syntax tree – which we will describe further in §5.3.2. However, the intermediate representation also depends on several other data structures, *e.g.*, abstractions of symbols and types. This section describes all the internal data structures that are used in the intermediate code representation in Fcc.

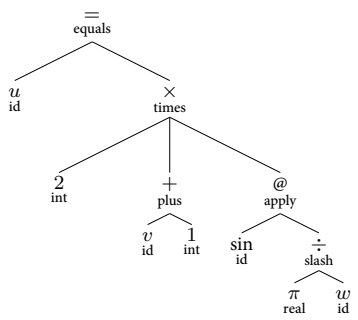
5.3.1 Abstract Syntax Tree

The abstract syntax tree is a unique representation of the input and takes the form of an n-ary tree. The intermediate code is stored in a dictionary of function definitions, and since each definition can consist of multiple function clauses, every function definition is in fact stored as an array of function nodes. The function bodies are stored in a format similar to the one that is used in Figure 5.3.

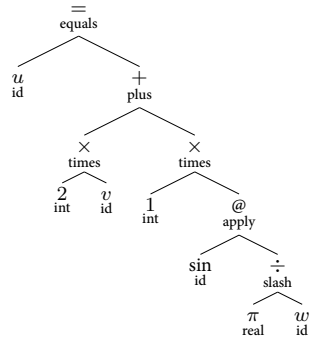
In the Python implementation of the compiler, each node of the syntax tree is an instance of the `Node`-class. Every `Node`-object holds a reference to a derived instance of the `DataNode`-class, *e.g.*, a `Number`-, `Operation`- or `SymbolReference`-object. This allows for easy book-keeping of node parents and for inline substitution of nodes and branches, as we will see in later sections. The justification for this architectural decision is that we want to simplify the tree manipulations that are carried out by the optimiser, see §5.3.5 on page 43.

5.3.2 Three-Address Code

In Fcc, the intermediate representation needs to take the form of three-address code (§2.4.1 on page 17) in some stages of the optimisation process to ensure the applicability of certain compiler transformations. TAC is represented in the form of an abstract syntax tree with explicit constraints imposed on the number of branches per node and on the structure of the tree. More specifically, the rules of the three-address code representation imply that all `Operation`-nodes only have two operands, and that both of these operands are on a trivial form, *i.e.*, in the form of an immediate value or a symbol



Input: $u = 2 \times (v + 1) \times \sin(\frac{\pi}{w})$.



Input: $u = 2 \times v + 1 \times \sin(\frac{\pi}{w})$.

Figure 5.3: The abstract syntax trees that are depicted above, illustrate the results of the syntactic analyses of the inputs printed below each tree. Note the effects of bracketing and the operator precedence rules that are defined in §A.2 on page 85.

reference. Consequently, the abstract syntax tree and three-address code representations are equivalent data structures, meaning that both can be fed directly into the type inference system without any alterations. The conversion of abstract syntax trees into three-address code is covered in §5.8 on page 48.

Since Fcc disallows destructive assignments, the employed intermediate representation, three-address code, is essentially equivalent to SSA. This gives us a good foundation for performing data-flow analyses. Other intermediate representations, such as the ones that are described in §2.4.1 on page 17, are not used by the Fcc compiler.

With respect to the implemented compiler transformations, the employment of, for instance, the continuation-passing style would not have had any direct effect on the optimisation process as the intermediate code is already in an equivalent format. Also, the fact that function inlining requires renormalisation of function bodies when representations other than CPS are used can be disregarded as the nature of the implemented abstract syntax tree structure would have required additional steps to be taken even if the continuation-passing style was employed. Regardless, in further work, one should consider to employ the continuation-passing style, as it could potentially enable the application of other, hitherto uncovered interprocedural transformations.

It is worth noting that, in intermediate stages of the compilation, the internal representation of the input code may contain destructive assignments, *e.g.*, in the internal representations of looping constructs. These are dealt with explicitly by the use of locks on symbols and instructions. The locks impose a set of constraints on the symbols in question, and these constraints are used to disallow certain transformations on the input during optimisation.

5.3.3 Types

As we will see in later sections, each node of the abstract syntax tree gets assigned a type based on the results of the type inference process. To be able to typify the nodes in the syntax tree, we need an explicit representation of all the types that are available in Fcc. Considering the types that are available to the programmer (see §A.1 on page 83), the following type attributes have been identified:

- Every type is either a scalar ($\mathbb{Z}, \mathbb{R}$), a vector ($\mathbb{Z}^*, \mathbb{R}^*$) or a function type.
- Scalar types and vector types are either integral ($\mathbb{Z}, \mathbb{Z}^*$) or real ($\mathbb{R}, \mathbb{R}^*$).
- Vector types are immutable and homogeneous; the latter meaning that vectors cannot mix integer elements with reals, and vice versa.
- Function types are recursive structures that are constructed from the above base types. More specifically, function types have zero or more parameter types, and a single return type, *e.g.*, $\mathbb{Z} \longrightarrow \mathbb{Z}$ and $\mathbb{Z}^* \times (\mathbb{Z} \longrightarrow \mathbb{R}) \longrightarrow \mathbb{R}^*$

These attributes must be visible in the internal representation of the types.

The type inference process capitalises on the fact that some types take precedence over others. The precedence of a type is a static measure that gets assigned to each type upon instantiation. More specifically, the types are ranked as follows:

$$1^1 < \mathbb{Z} < \mathbb{R} < \mathbb{Z}^* < \mathbb{R}^* < \mathcal{F} \qquad \text{where } \mathcal{F} \text{ denotes function types.} \tag{5.1}$$

The types that are listed in (5.1) are represented by instances of the `Type`-class, which is defined in "`semantics/types.py`." All the base types are statically defined, whereas function types are composed at run-time from the deduction of implicitly typed function signatures.

The `type`-attribute of the nodes in the abstract syntax tree is a dynamic attribute which deduces the type of the node upon invocation. The type is deduced based on the data of the `Node`-instance in addition to a set of constraints that can be imposed by the `typify`-method. As described in §5.5 on page 44, the combination of calls to `type` and `typify` is used by the type inference system to stabilise the typing of the input program. For classes such as `Operation`, whose type depends on an arbitrary number of sub-nodes, the `Type.dominion` class-method provides functionality to deduce the most dominant type from a set of types, *i.e.*, the type of the highest precedence.

5.3.4 Symbols and Environments

All identifiers that are found in the input program during the semantic analysis get stored in a symbol table so that each symbol can be referenced uniquely by a memory pointer in later stages of the compilation. The use of a symbol table reduces the number of strings comparisons that are needed to perform a symbol lookup. Furthermore, data duplication is eliminated as the symbol table ensures unicity of symbols and provides us with a way of storing additional information about each entry, *e.g.*, information about the type, scope and origin of each symbol. The symbol table and all the functionality that is relevant to the symbol table implementation can be found in the `Symbol`- and `SymbolReference`-classes in "`semantics/tree.py`."

Scope levels are trivially defined in Fcc. All functions reside in the global scope, and all variables and parameters reside in the parent function's local scope. The intermediate code does in fact use additional scope levels when dealing with variables inside looping constructs. However, these are all distinct and invisible to the programmer. Since Fcc has no constructs for lexical scoping and due to the fact that the language has single assignments, the need for additional scope levels is inexistent.

[1] 1 denotes `void`, a type which is commonly used to represent the set of parameters for nullary functions, *e.g.*, $1 \longrightarrow \mathbb{Z}$ is a function type with no parameters and an integral return type.

5.3.5 Change Propagation

During the optimisation phase, certain branches and leaves of the syntax tree may be subject to alteration. Changes that are made to these nodes naturally propagate up the tree and may cause other simplifications and alterations to be exploitable at higher levels.

For example, consider the n-ary syntax tree:

```
(4 * 2) + X + 2  ↦  add(mul(int(4), int(2)), id(X) int(2)).    (5.2)
```

If we transform `mul(int(4), int(2))` into `mul(int(8))` and perform an inline substitution to get `int(8)`, we end up with the tree `add(int(8), id(X), int(2))`. However, we would now like to automatically update the parent node so that the recent changes can be exploited at higher levels of the tree. To avoid having to reprocess the entire tree for every little change, an *update* event gets invoked on the changed node's parent. More specifically, the `update`-method of the top-level `Operation`-node gets invoked so that the appropriate optimisations can be carried out. In Fcc, the `update`-method for `Operation`-nodes performs constant folding, amongst other transformations. So in this scenario, the invocation will yield the expected result, `add(int(10), id(X))`.

Every node of the syntax tree implements an `update`-method, and this dynamic mechanism is extensively used in the type inference and optimisation phases of the compilation to propagate all changes up the tree and to ensure a coherent structure. Note that the triggering of events will propagate all the way up to the root node. Hence, the depth of the tree is insignificant.

The separation between the `Node`-class and the `DataNode`-class is in fact a result of the design that is described above as it allows for easy substitution of nodes and easy book-keeping of parents. In the current implementation, inline substitution is simply a matter of changing the value of the `data`-attribute of the `Node`-object. Whereas, if the separation between the two classes had not been in place, we would have had to keep an explicit pointer to one of several slots in the parent node, namely the slot that was used to store a reference to the considered node object. The latter structure complicates the process of keeping a coherent tree structure. Furthermore, Python does not support multiple levels of indirection and, thus, the naïve design would have required an extra level of abstraction.

We did in fact see an instance of inline substitution in the example above, *i.e.*, the transformation of `mul(int(8))` into `int(8)`. The first instance is a shorthand for the Python structure:

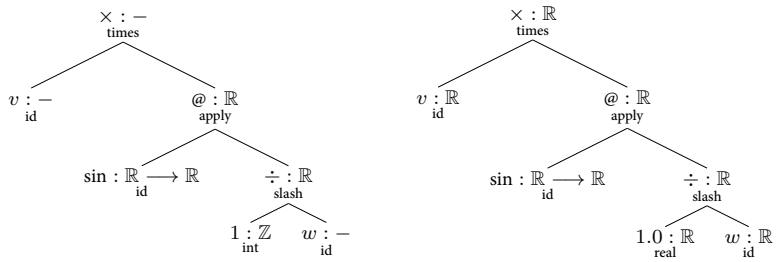
```
Node(Operation('*', Node(Number(8)))),
```

which, as a result of the inline substitution that is described above, gets restructured into `Node(Number(8))`. More specifically, the `data`-attribute of the outer node-instance gets changed to point to the instance of the `Number`-class instead of the `Operation`-class.

5.4 Feedback

As stated in Chapter 3 on page 28, the compiler must provide feedback to the user on the results of the compilation. The compiler must also print error messages if the provided input program does not comply with the language specification, and warning messages if the program contains unused symbols or unreachable code.

To simplify error recovery and elimination of unused symbols and unreachable code, the compiler keeps track of the exact source code location of all the tokens in the input



Raw Data from the Syntactic Analysis

Abstract Syntax Tree After Type Deduction

Figure 5.4: This figure illustrates the results of a type deduction that has been carried out on the left-hand side syntax tree. Observe how the type of the sin-function and the resulting type of the real division propagate up and down the tree. Observe also how integers are converted into reals when type constraints are added.

program. The locations are associated with the nodes in the syntax tree, and take the form of a `Location`-object. The `Location`-class provides additional functionality that allows for easy printing of formatted error and warning messages to the standard error stream.

Fcc conveys extensive error and recovery information to the end-user. In addition to providing messages about syntax errors and missing entry-points, the compiler informs the user about the following problems:

- Unterminated function groups, overlapping signatures for pattern matching function clauses, and missing catch-all clauses (§A.5 on page 87).
- Invalid operand types, and invalid argument types in function applications.
- Invalid number of arguments in function applications.
- Inconsistency in the return types of function clauses of a function group.
- Duplication of parameters, and destructive assignments to variables.
- References to undefined variables and functions.

The compiler outputs warning messages if the input program contains any unused variables or functions, or if the type inference process fails to deduce a stable type system.

5.5 Type Inference

In §5.3.3 on page 41, we defined the type system of Fcc. The type inference component of the compiler is a part of the semantic analyser, and takes an iterative approach to deduce the type of all the variables, functions and parameters in the input program. If no stable solution of typing can be deduced, the process fails, *i.e.*, no valid output program can be generated, and the user gets an appropriate notification describing the problem and any inconsistencies that may exist.

As mentioned in previous sections, the type inference system exploits the functionality that is provided by the `type-` and `typify`-methods of the `Node`-class. These methods are virtual and end up redirecting the caller to the corresponding methods of the

interlinked `DataNode`-objects. The `type`-method is used to deduce the current type of a branch of the syntax tree, whereas `typify` is used to add typing constraints.

By iterative application of these methods, the compiler eventually reaches a stable state, *i.e.*, a state where the type of none of the nodes has changed as a result of the current iteration. However, on bad input, such a process has the potential to never terminate. Thus, a threshold has been implemented so that the process is guaranteed to stop if a stable solution has not been reached in a reasonable number of steps.

The type deduction process complies with the type model that is specified in §C.1 on page 97. The general principle is that all the leaf nodes have a type associated with them, except from symbols. *E.g.*, integers are obviously of type $\mathbb{Z}$, and reals of type $\mathbb{R}$. Symbols are typed based on their parent nodes. For instance, the symbol w in Figure 5.4 will obtain the type $\mathbb{R}$ since real divisions always yield quotients of type real. Such type assignments will propagate to all references to the involved variables, here w, by the invocation of the `typify`-method.

The introduction of type constraints to immediate values yields type conversion if the new type has higher precedence than the existing one. Note how the changes that are caused by the type inference system are propagated in both directions of the tree – changes in a higher level of the tree will propagate down to the leaves and changes in the leaves will propagate up to the root.

5.6 Live Variable Analysis

Dead variable elimination, copy propagation and other compiler transformations are dependent on having knowledge about the liveness of variables in the currently processed basic block. Recall that a basic block is a branch-free sequence of three-address code instructions of the form $d \leftarrow o_1 \bullet o_2$. Algorithm 5.1 describes the employed liveness analysis in general. The algorithm returns a collection of sets, each of which holds the live variables at a certain execution point of the program, *i.e.*, L_i is the set of variables that are live upon the execution of the instruction, t_i.

Algorithm 5.1 Live Variable Analysis – based on Algorithm 8.7 in [ALSU06].

Input: A basic block, B, of three-address code instructions t_i, $1 \leq i \leq n$.
Output: $L = \{L_1, \ldots, L_n\}$, where L_i is the set of live variables after execution of t_i.

1: $LV \leftarrow \{\}$
2: for $i = n$ down to 1 do
3: {Assume t_i is of the form $x \leftarrow y \bullet z$.}
4: $L_i \leftarrow LV$
5: $LV \leftarrow LV \setminus \{x\}$
6: $LV \leftarrow LV \cup \{y, z\}$
7: end for
8: return L

The liveness analysis is extensively used in the optimisation and parallelisation phases. Since FCC implements single assignments, the implementation of the liveness analysis takes a slightly different approach to check whether a variable is live or not. Single assignments imply that there are no destructive assignments to variables. Thus,

checking for liveness is simply a matter of checking whether or not the considered variable is referenced at a later stage of the basic block[2].

Generally, we ought to consider destructive writes as they can result in seemingly live variables actually being dead, *e.g.*, consider a variable between a read in t_k and a destructive write in t_l, where $l > k$. However, in Fcc, this criterion lapses. Also, in this context, it suffices to test for liveness on a basic block level as there are no constructs for lexical scoping, *i.e.*, a function body is essentially a single basic block. In intermediate stages of the compilation, scoping and separation between basic blocks occur if an instance of recursion is converted into iteration. The compiler then tests for liveness across basic blocks and will have to consider destructive writes to all iteration variables.

5.7 Dependence Analysis

This section presents the dependence analyses that have been implemented in Fcc and which are used by the compiler during the parallelisation phase.

In §2.4.6 on page 22, we presented a set of control- and data-dependences that are commonly found in computer programs. By identifying such dependences, we can determine an optimal execution order of instructions, with the objective to, for instance, improve the cache locality in a code block. However, in Fcc, dependences are analysed only to ensure that there are no loop-carried dependences in the loops that are selected as candidates for parallelisation.

Algorithm 5.2 Loop-Carried Dependence Analysis

Input: A basic block, B, of three-address code instructions t_i, $1 \leq i \leq n$, representing the body of a parallel `for`-loop, and a variable H representing the currently processed element of the input list, L.

Output: True, if the loop body carries dependences across iterations, false, otherwise.

1: { Input loop is of the form: `for-each (`$H \in L$`) { ... }`. }
2: $R \leftarrow \{\}, W \leftarrow \{H\}, P \leftarrow \{\}$
3: for $i = 1$ to n do
4: { Assume t_i is of the form $x \leftarrow y \bullet z$, and that x, y, z are variables. }
5: if operation ($\bullet$) is *list prefixation* then
6: {Previous analysis ensures that t_i is of the form H `=` y, meaning that t_i is an update of the currently processed element of L. }
7: if $y \notin W$ then
8: $P \leftarrow P \cup \{y\}$ { Not previously written in the current iteration. }
9: else
10: if $y \notin W$ then $P \leftarrow P \cup \{y\}$
11: if $z \notin W$ then $P \leftarrow P \cup \{z\}$
12: end if
13: $R \leftarrow R \cup \{y, z\}, W \leftarrow W \cup \{x\}$
14: end for
15: return $(W \cap P) \neq \emptyset$

As mentioned in previous sections, the objective of the compiler is to identify and exploit parallelisable loops. Since the identified candidate loops get parallelised for execution on data-parallel architectures, the compiler focuses on the parallelisation of

[2]Note that the inclusion of internally represented looping constructs in basic blocks is invalid and results in a splitting of the original blocks.

mapping- and reduction-style problems. Fcc uses dependence analysis to identify candidates for parallelisation, *i.e.*, loops without loop-carried dependences, and afterwards, uses the criteria that are listed later in this section to identify well-behaved reduction and mapping constructs.

In Algorithm 5.2, we see how Fcc searches for interdependences between loop iterations. The compiler explicitly checks that read accesses to variables only happen when the variables in question have been written to earlier in the current iteration of the loop, or not at all (which means that we are dealing with a global variable). Due to the nature of the transformations from recursion to iteration, and due to the constraints that are imposed on us by the supported data structures, updates to elements at, say, index j for a currently processed element i, where $i \neq j$, are covered by the more general dependence checks in the algorithm, *i.e.*, we need not consider the invalidation technique that was mentioned in §2.4.6.3 on page 23.

To check whether a candidate loop is a mapping operation, a reduction operation, or potentially neither of the two, is a fairly straightforward procedure. To identify mapping operations, the compiler checks the following criteria:

- The considered loop must be identified as a parallel `for`-loop, meaning that the original, recursive function must be on a well-defined, inductive form with a clearly distinguishable base case, the empty list, and a single inductive step mapping the head element of the processed list. This criterion is a result of the process of converting recursion into iteration – the conversion is described in more detail in §5.9.3 on page 51.
- The return type of the original recursive function must be a vector type, *i.e.*, $\mathbb{Z}^*$ or $\mathbb{R}^*$. This follows from the previous criterion.
- Exactly one of the instructions in the loop body must process the head element of the input list and reset its value based on the computational result.

To identify reduction operations, the compiler checks the following criteria:

- The considered loop must, as before, be identified as a parallel `for`-loop, meaning that the structural recursion must be on a well-defined, inductive form with a single base case, the empty list, yielding a result of type integer or real, and exactly one inductive step.
- There must be no direct operations on the traversed input list. The only valid operations are read accesses to the current head element of the induction variable, *i.e.*, the element of the list that is currently being processed.
- The result variable, which is identified from the recursion-to-iteration conversion, can only be assigned to once. In other words, the loop body must hold exactly one assignment on one of the following forms – where P is the result variable:

  $P \leftarrow P + t$ or $P \leftarrow P \times t$ where t cannot hold any references to P.

  Consequently, Fcc only deals with additive and multiplicative reduction.

5.8 Elementary Compiler Transformations

The optimisation phase of the compilation is an iterative process. This is a result of the fact that certain transformations enable others, meaning that some derivations can only

be found after a set of other transformations has been performed on the input. The reason why we choose to take an iterative approach is to be able to stabilise[3] this process and, hence, be able to derive the optimal solution to the optimisation problem.

The first step of the optimisation process is to ensure that the intermediate representation of the input program, meaning the abstract syntax tree, is in a format that satisfies the rules for three-address code – *cf.* §5.3 on page 40. If the abstract syntax tree does not already comply with these rules, we can easily convert it into a valid format by applying Algorithm 5.3 to all expression branches of the tree. The output of this transformation may contain excessive code or use unnecessarily many temporary variables. However, this is handled by the optimiser in later stages. It should be noted that Algorithm 5.3 only provides an outline of the translation process, and that there are special cases that need more careful handling. For instance, operations are represented as n-ary nodes, not binary nodes, and need to be split into multiple assignments of binary operations into temporary variables. The full implementation of the algorithm can be found in "`semantics/ssa.py`."

Algorithm 5.3 Translation into TAC – based on Figure 2.45 in [ALSU06].

Input: Expression, x, in the form of a branch of the currently processed syntax tree, a code block, C, in which the expression resides (more specifically, a `Combinator`-object), and the index, i, of the statement holding the expression.

Output: Expression satisfying the axioms of three-address code. Note that the code block, C, may experience modifications as a result of the translation.

1: if x is an immediate value or a symbol reference then
2: return x {Trivial operand, no action needed.}
3: else
4: {The input, x, is guaranteed to be of the form $y \bullet z$.}
5: Generate new temporary, t.
6: $y' \leftarrow$ Algorithm 5.3(y)
7: $z' \leftarrow$ Algorithm 5.3(z)
8: Create and insert new branch, $t \leftarrow y' \bullet z'$, at index $(i - 1)$ in C.
9: return t
10: end if

Once the abstract syntax tree is in a three-address code format, we can apply a set of elementary compiler transformations to the tree, more specifically, the commonly applied compiler transformations that we described in detail in §2.4.2 on page 19. Some of these transformations depend on the liveness analysis that was described in §5.6.

In Fcc, we use an extended form of three-address code, and the reason for this is that we want to exploit the additional functionality that is provided by the abstract syntax tree structure. Strictly speaking, constant folding, algebraic transformation and strength reduction all get carried out by the `update`-method of the `Operation`-class. Nevertheless, these transformations are still part of the iterative optimisation process that was outlined above.

In contrast to the transformations that are enabled by the abstract syntax tree structure, some of the transformations must be dealt with elsewhere, by other means. These are handled by the `Optimiser`-class, whose implementation can be found in "`optimisation/general.py`," and are listed below:

[3] By stabilising an iterative process, we mean reaching a state where no more transformations can be applied, *i.e.*, no further changes can be made to improve the performance of the program.

- Common Subexpression Elimination – This transformation substitutes duplicate subexpressions with the assignment destination of the first occurrence.
- Copy Propagation – This transformation propagates s_1 from all instructions of the form $d \leftarrow s_1$ to all operands that are equal to d in subsequent instructions, unless the data-flow is interrupted by a destructive assignment to d. Destructive assignments are inexistent in FCC. Nevertheless, they can occur in the internal representation, *e.g.*, in the representation of looping constructs where iteration can cause a seemingly well-behaved assignment to become destructive.
- Dead Variable Elimination – This transformation eliminates all instructions, t_i, of the form $d \leftarrow s_1 \bullet s_2$, where d is dead after the instruction, *i.e.*, $d \notin L_j$ for all $j \geq i$ (Algorithm 5.1).

After the first phase of the optimisation is completed, meaning when the iterative application of the compiler transformations that are listed in this section has stabilised (as illustrated in Figure 5.1), the compiler rebuilds trivial expressions from the three-address code. This means that the compiler propagates the right-hand side of all assignments of the form $d \leftarrow s_1 \bullet s_2$, where d is only referenced once in the subsequent instructions, to the rest of the instructions in the containing code block and eliminates the single-referenced assignments altogether. This reduces the overhead that is imposed on us by superfluous use of temporary variables – even though the use of such additional variables is likely to be eliminated by the external build tools. Furthermore, the rebuilding of expressions can simplify the semantic reasoning in later stages of the compilation process, *e.g.*, in the interprocedural optimisations and in the recursion-to-iteration conversions.

Since FCC is a pure functional language, function calls are referentially transparent. Thus, after having completed phase one of the optimisation, the compiler also collects all equivalent function invocations residing in the same scope, and substitutes these by references to a temporary variable which has been assigned the result of a single invocation of the function. For instance, $y \leftarrow f(g(x), g(x))$ becomes: $t \leftarrow g(x);\ y \leftarrow f(t, t)$.

The final step before the optimiser moves on to phase two is to clean up the intermediate representation. This step entails the removal of unused symbols and functions. It is worth observing that phase one can be revisited after the second phase has been completed. This happens if the intermediate representation changes as a result of the current iteration of the optimisation process – in fact, this would then also involve a reiteration of the type inference process.

5.9 Interprocedural Optimisations

The first phase of the optimisation process deals with trivial and commonly applied compiler transformations. In this work, the application of these transformations has the objective of simplifying the semantic analysis that is employed in later stages of the compilation. The process of making the input amenable to automatic parallelisation is heavily dependent on the interprocedural optimisation techniques. More specifically, as we have mentioned in previous sections, the parallelisation of pure, functional code for data-parallel architectures is first and foremost a matter of converting recursion into iteration, in the form of data-parallel loops.

5.9.1 Procedure Cloning

Fcc implements a dynamic model for determining whether a loop should be run sequentially or in parallel. Hence, the compiler needs to generate multiple versions of the same function. To accomplish this, the procedure cloning transformation is employed.

Procedure cloning is an optimisation technique which creates specialised replicas of the original function, and which updates the linked call-sites accordingly [CHK92]. In the case of Fcc, two specialised copies are made, one for sequential execution and one for parallel execution. During function replication, all local variables and parameters are duplicated and re-referenced so that there are no links between the new and the old copy of the function. The dynamic model, determining which of the two specialised function copies should be run, is described in §5.12.2 on page 58.

5.9.2 Procedure Inlining

In purely functional programming languages, the functionality of the implementation tends to be distributed across multiple functions, each with a relatively small body of statements. Consequently, functional-language compilers are dependent on interprocedural optimisation techniques to be able to produce high-performance programs. This is a contrast to imperative languages, in which we often find large function bodies with multiple looping constructs, *etc.*, and whose compilers are predominantly concerned with the intra-procedural optimisation techniques.

Procedure inlining is perhaps one of the most important interprocedural optimisation techniques for functional languages. This technique can substantially reduce the overhead that is associated with function invocations, as it reduces the overall stack usage of the program [BGS94]. Inlining of functions can introduce new local variables, which essentially reside in the run-time stack. However, most compilers try to map local variables onto the registers that are available on the user's machine [ALSU06], and since the latency of accessing registers is much lower than the latency of accessing memory, superfluous use of the stack is considered more costly than the use of local variables in general. Furthermore, procedure inlining can enable other optimisations and improve the quality of the semantic analysis that is employed in the conversion of recursive functions into loops.

In the context of this work, it should be noted that the reduced stack usage also implies better compatibility with the target GPU architecture. Generally, GPUs have limited stack support due to the delicate memory architectures of these devices. Therefore, GPU compilers, such as the CUDA compiler, often disallow recursion and inline functions that are targeting the graphics device, by default [NVI10b, Mun10].

Not all functions can be inlined, *e.g.*, consider infinitely recursive functions. Therefore, we need to check whether the candidates for procedure inlining satisfy certain criteria. Inlining is a multi-step procedure where the compiler considers all the function applications in the input program and performs the following steps:

- First, the compiler checks whether or not the target of the function application is a trivial leaf function. For the function to be a trivial leaf, the following conditions must be satisfied:
  - There must be no function applications in the function body – exceptions include the special-purpose functions that are listed in §A.3 on page 85.
  - The function must not be referenced as a first-class function in other parts of the program.

  - The function signature must be trivial, *i.e.*, there must be no pattern matching on any of the function arguments.
  - The condition-clause of the function must be blank[4].

- If the above criteria are met, the compiler clones the target function of the call-site and retains a copy.
- The compiler then substitutes all parameter references in the cloned function with their corresponding arguments from the function application.
- Finally, the compiler substitutes the function application node with the body of the derived function.

Upon completion of the inlining process, there will no longer be any references to trivial leaf functions in the program. Trivial leaf functions can therefore be omitted from the generated output and, consequently, removed from the intermediate representation.

5.9.3 Optimisation of Augmenting Recursion

The prime optimisation candidates in Fcc are functions that implement augmenting recursion – see definitions, algorithms and proofs in §B.1 on page 90. The conversion of augmenting recursion into tail recursion has been discussed on multiple occasions in the literature, see for instance [LS00, BD77], and has proven to be an important and beneficial transformation in functional-style programs. However, most compilers tend not to implement this transformation as an automatic transformation and, therefore, leave it up to the programmers to manually rephrase their functions to exploit the advantages that are associated with the employment of tail recursion.

Listing 5.1: This listing shows how the classic factorial function can be implemented in Fcc. Note that the function employs augmenting recursion to accomplish its objective.

```
fact(0): 1;                          # base case
fact(N): N * fact(N-1).              # recursive case
```

Fcc automatically transforms augmenting recursion into tail recursion. This yields improved performance in sequential programs as it enables the conversion of recursion into iteration and, hence, has the potential to completely annihilate the linear stack growth that is associated with conventional recursion – see §5.9.4.

Listing 5.2: The code in Listing 5.1 can be optimised by converting the instance of augmenting recursion into tail recursion, see §B.1.1 on page 92.

```
fact(N, T, R), N == T: N*R;          # a(0)     = 1
fact(N, T, R): fact(N+1, T, N*R).    # g(A, B) = A * B
fact(0):       1;                    # d(N)     = N - 1
fact(N):       fact(1, N, 1).        # d^-1(N)  = N + 1
```

[4]The built-in `if`-function is eagerly evaluated as a result of the design of the Fcc language. Since a conditional would imply lazy evaluation of the function body, and furthermore, since there exists no equivalent representation in the Fcc intermediate representation, such a function cannot be inlined unless the necessary internal constructs are added to the compiler implementation.

However, this is not the only benefit of optimising instances of augmenting recursion. Observe that instances of structural recursion essentially get converted into iterative constructs of the form of a parallel `for`-loop. Thus, the transformation of augmenting recursion into tail recursion and iteration also enables the parallelisation of structural recursion for data-parallel architectures, such as CUDA.

Generally, it is often easier and cleaner to express structural recursion in the form of augmenting recursion than in the form of tail recursion, as shown in §B.1.4 on page 94 and discussed briefly in §2.4.5 on page 21. Consequently, the conversion of structural recursion in the form of augmenting recursion into tail-recursive equivalents, is a crucial task for the FCC compiler. This becomes evident when we consider the potential size of the processed data structures. The processed lists can be large and cause a huge stack overhead. Thus, it is vital that we are able to derive iterative versions of the recursive functions. It is also worth noting that, by sticking to the original function representations which implement augmenting recursion, we might not even be able to compute the answers as we might run out of stack space before the answers have been deduced.

Listing 5.3: The code in Listing 5.2 can be further improved since g, the multiplication of two integers, is an associative and commutative function. The following snippet shows the result of the extended optimisation, which is described in detail in §B.1.2.

```
fact(0, R): R;                  # base case
fact(N, R): fact(N-1, R*N).     # recursive case
fact(N):    fact(N, 1).         # helper function
```

The code listings that are provided in this section show an example of how the FCC compiler transforms instances of augmenting recursion into tail recursion. The last step of this transformation, the step that, here, exploits the associativity and commutativity of the identified function, g, and whose result is exemplified in Listing 5.3, is not necessarily beneficial in terms of performance. In the case of the factorial function, the last step would actually turn a multiplicative sequence of the form:

$$f_n \times (f_{(n-1)} \times (f_{(n-2)} \times \ldots (f_1 \times f_0) \ldots)), \tag{5.3}$$

into an equivalent sequence of the form:

$$((\ldots((f_n \times f_{(n-1)}) \times f_{(n-2)}) \times \ldots) \times f_1) \times f_0, \tag{5.4}$$

where $f_n > \ldots > f_1 > f_0$. The computation of (5.4) would require more large-number multiplications than the computation of (5.3), and would therefore be substantially more costly to compute, performance-wise, especially for large values of n [LS00]. In other words, the order in which the function, g, is applied, can affect the running time of the program. This is clearly demonstrated in the example above, where we would normally opt for the former alternative, *i.e.*, the version of the factorial function which is presented in Listing 5.2. A natural counter-example is list reversal [LS00], which is an instance of structural recursion that can be expressed in the form of augmenting recursion. In further work, an extended performance model should be employed to automatically detect whether or not the last step of the optimisation of augmenting recursion should be performed on applicable functions[5].

[5]The last step of the transformation is only applicable to functions that implement augmenting recursion and where the identified g-function is associative and commutative.

5.9.4 Tail-Call Elimination

If the last statement of a function body contains a function call whose return value gets returned directly without any other interference, the function is said to have a tail call. We can exploit the fact that the target functions of such calls do not have to return to the call-site by substituting the function call with a jump instruction. This allows us to reuse the current stack frame instead of setting up a new activation record and allocating additional stack space for the function arguments [Ste77, BGS94].

Functions with recursive tail calls are called tail-recursive functions. These functions can benefit even further from the technique that is outlined above, since they can be turned into looping constructs where the bodies of the original functions form the bodies of the generated loops, and the function parameters constitute the associated iteration variables. The concept of converting recursion into iteration can be extended to deal with mutual recursion [LS00]. Fcc can transform tail recursive functions with one or more base cases into iterative constructs. However, the compiler cannot deal with instances that have more than one recursive call.

Listing 5.4: The classic factorial function can be optimised by converting augmenting recursion into tail recursion (Listing 5.3). The function can also be optimised further by eliminating the tail call and turning the function into a loop.

```
fact(N, R): while (N > 0) { R′ = R*N, N′ = N-1, R = R′, N = N′ }, R.
fact(0):    1;
fact(N):    fact(N-1, N).

# applicable to procedure inlining, which yields:
fact(N): R = 1, while (N > 0) { R = R*N, N = N-1 }, R.
# the intermediate use of R′ and N′ is eliminated by the optimiser.
```

In addition to preparing the input for parallelisation by simplifying the compiler's job in identifying cases where parallelisation is possible, the combination of converting augmenting recursion into tail recursion and tail-call elimination annihilates stack growth. Thus, the transformation of recursion into iteration can also act as an enabler for traversals over huge data-sets, *i.e.*, in cases where the program would normally run out of stack space because of the depth of the recursion.

5.10 Loop Optimisations

There exist no explicit looping constructs in the Fcc language. Nevertheless, the previously described compiler transformations have the potential of generating intermediate constructs for iteration, *e.g.*, tail-call elimination can turn a recursive function into a loop. These intermediate constructs should always be subjected to optimisation as loop transformations constitute an important part of the optimisation process for data-parallel architectures [BGS94].

The current version of the Fcc language only supports one-dimensional, homogeneous arrays. Due to this limitation, only a few of the loop transformations that are commonly discussed in the literature, are applicable to the compilation process of Fcc. For instance, the current version of the compiler will never end up in a state where it produces nested loops, so it would make no sense to try to apply transformations such as loop coalescing, loop collapsing, loop tiling, loop interchange, *etc.*

Another common loop transformation which is not implemented in Fcc, is loop unrolling. The CUDA compiler unrolls amenable loops by default, and so, Fcc could be made to rely on this feature. However, the compiler is unable to produce loop nests due to the single-dimensionality of the supported data structures and the constraints that are placed on procedure inlining and the recursion-to-iteration transformation. Loop nests are essential to the employment of loop unrolling as the outer loops of nests are turned into GPU kernels, meaning that if we only have a single level of nesting, the entire loop is made into a kernel and the loop will disappear. Thus, the benefits from employing loop unrolling cannot be exploited in the current version, and the need for the unrolling feature of CUDA lapses.

Some compilers also exploit a combination of loop unrolling and loop rerolling to improve the quality of the other loop optimisations [BGS94]. The implementation of this feature lapses of the same reasons as those that are listed above.

5.10.1 Loop Fusion

Fcc is, as we have seen previously, unable to produce loop nests because of the single-dimensionality of lists and the constraints that are imposed on procedure inlining (§5.9.2 on page 50) and on the optimisations that are applied to recursive functions (§5.9.3 on page 51). However, the compiler has the capability of producing a set of consecutive loops within a single function body. Consecutive loops that are traversing over the same data are likely to introduce additional overhead due to the performance penalties that are associated with looping[6]. Thus, we want to merge such loops to improve the overall performance of the program – a transformation called loop fusion.

From a parallelisation perspective, loop fusion allows us to invoke a single GPU kernel instead of multiple ones. This improves the performance of the compiled program as the instantiation and execution of GPU kernels, not to mention the associated memory transfers between the CPU and the GPU, can be quite costly and should be kept to a bare minimum. The loop fusion transformation is implemented in Fcc to attack the latter issue of superfluous kernel invocations.

Loop fusion can also be implemented to target loops that are generated from instances of non-structural recursion. However, this may require unification of the boundaries of the iteration variables. Non-structural recursion is not parallelised by the Fcc compiler and, thus, the merging of such loops is ignored in this work.

5.10.2 Loop Restructuring

Tail-call elimination produces raw `while`-loops from tail-recursive functions. To simplify the analysis that is employed in the parallelisation stage, these raw loops are attempted transformed into sequential and parallel `for`-loops. The compiler tries to identify instances of well-behaved induction from the operations that are applied on the induction variables, as described in §5.7 on page 46. To improve the quality of this analysis, the compiler utilises the fact that Fcc implements pattern matching on function arguments to help identifying the core of the induction. The latter helps filtering out seeming induction variables which are used to accumulate the results of the inductions instead. The complexity of the generated loops depends on how many parameters there are in

[6]The condition checks that are performed for every iteration of the loop and the code for progressing to the next element of the traversed list can be costly in terms of performance when we are iterating over huge data-sets.

the original functions, and on how many of these parameters remain static throughout the execution of the loops.

Strictly speaking, the generation of sequential `for`-loops does not affect the parallelisation stage of the current version of the compiler. However, this transformation can prove important in later work, as it enables the application of further loop transformations and, thus, has the potential to improve the data locality of the generated loops. This is particularly beneficial if the compiler produces loop nests and if the outer loops of these nests get parallelised. The parallelisation of such loop nests would yield GPU kernels that execute sequential loops, and it would be desirable for these sequential loops to access data in predictable ways.

5.10.3 Loop Idiom Recognition

The optimiser tries to identify well-behaved instances of induction, a process which is based on the analysis that was outlined in §5.7 on page 46. All candidates are either of the form of parallel mapping or parallel reduction, and are flagged as such so that they can be exploited during the code generation phase.

5.11 Automatic Parallelisation

As is clearly stated in the objectives that are listed in §1.4 on page 3, Fcc focuses on the pre-parallelisation phases of the compilation of functional-style code. Consequently, the compilation process is mainly concerned with the type inference process and with the compiler transformations that have been discussed up until now.

Even if the focus of this work has been put into preparing functional-style code for parallelisation, the implemented prototype compiler also implements a component which parallelises applicable looping constructs – which have been derived from the application of the previously discussed optimisation techniques. This component targets the CUDA platform, and deals with the identification and translation of the following kinds of loops:

- Mapping Constructs – All parallel `for`-loops[7] that apply an operation or a function to each element of the traversed data structures. Note that the operation / function can only operate on the sole element that it has been given, and that no cross-referencing of elements can occur.
- Reduction Constructs – All sequential `for`-loops that compute the sums or the products of all the elements of the traversed data structures – such loops are called additive and multiplicative reduction constructs, respectively.

The two types of looping constructs that are described above are not mutually exclusive as, for instance, sequential `for`-loops that compute the sums or the products of some maps of the traversed data structures are equally valid candidates for parallelisation. The general criteria for parallelisation of such mapping and reduction constructs are given in §5.7 on page 46.

The fact that a candidate loop satisfies the criteria for being a mapping or reduction construct is not enough to guarantee that the parallelisation of that particular loop will yield an improved run-time performance. To guarantee speedup, we need to ensure

[7] Here, a parallel `for`-loop is a loop without loop-carried dependences. Conversely, a sequential `for`-loop is a loop with at least one identified loop-carried dependence.

that the size of the processed data and the workload of the loop make the parallelisation worthwhile, *i.e.*, that the list is sufficiently long and that the loop body performs a relatively costly operation on the elements, in comparison to the size of the data.

As we have seen in previous chapters, memory transfers between the host and the device can be costly, and they should therefore be kept to a minimum. To minimise the communication overhead that is imposed on us by parallelisation, the compiler tries to merge subsequently executed GPU kernels that are working on the same data. Furthermore, a parallelisation model, (5.7), is employed to rule out the cases where the cost of running a loop sequentially is lower than the cost of transferring the data to the device, invoking the GPU kernel, and transferring the result back to the host. To be able to give an estimate of the latter, the length of the traversed list must be taken into account. The size of the list is not always deducible at compile-time. Hence, a partially dynamic model must be employed.

Generally, the cost of a `for`-loop which traverses over a list, L, sequentially, can be modelled as:

$$\mathcal{P}_{\text{seq}}(L) = (\delta + \mathcal{P}[\![\texttt{body}]\!]) \times | L |, \tag{5.5}$$

where $| L |$ denotes the length of the list, δ is an estimate of the loop overhead, and $\mathcal{P}[\![\texttt{body}]\!]$ is the computational cost of the loop body. The parallel execution of the same `for`-loop, traversing over the same data, can be modelled as:

$$\mathcal{P}_{\text{par}}(L) = (| L | \times \gamma) + \mathcal{P}[\![\texttt{body}]\!] + \xi, \tag{5.6}$$

where γ is an estimate of the communication overhead that is caused by the memory transfer of a single list element, and ξ is the cost of a GPU kernel invocation.

Deciding which candidate loops to parallelise is then a matter of identifying loop instances where the size of the input list, L, satisfies:

$$\mathcal{P}_{\text{par}}(L) < \mathcal{P}_{\text{seq}}(L). \tag{5.7}$$

The performance models, (5.5) and (5.6), only provide an estimate of the actual workload of a loop. Hence, we cannot guarantee speedup by applying the models to the parallelisation process. In fact, candidate loops can occasionally be tagged for parallelisation even if the sequential versions of the loops run faster in reality. On the other hand, since the performance model is derived empirically from a model which is based on the costs of arithmetic operations on CUDA devices [NVI10b], the deviations in running time on these occasions can be disregarded as they are unlikely to contribute to a substantial slowdown of the execution of the output program.

The fact that the size of the traversed data structures need to be taken into account, implies that a part of the deduction of whether a loop should be run in parallel or not must happen at run-time. Thus, the compiler implements a hybrid between a static and a dynamic performance model. The static part of the model estimates the cost of the bodies of the candidate loops at compile-time, whilst the dynamic part is a set of minimalistic run-time checks which combines the pre-computed costs with the lengths of the traversed lists and branches to the desired loop version accordingly. Note that the compiler automatically generates two versions of each candidate loop, one for sequential execution and one for parallel execution. The complete model is described in detail in §C.2 on page 100. The implementation of the model and the CUDA code emitter can be found in "`analysis/complexity.py`" and "`backend/cuda/__init__.py`," respectively.

With respect to the parallelisation of reduction-constructs, it is worth noting that maximum parallelism can be achieved by computing the reduction in a tree-like fashion, where each branch of the computation tree corresponds to the summation or multiplication of two of the numbers of the list or of two of the previously computed results,

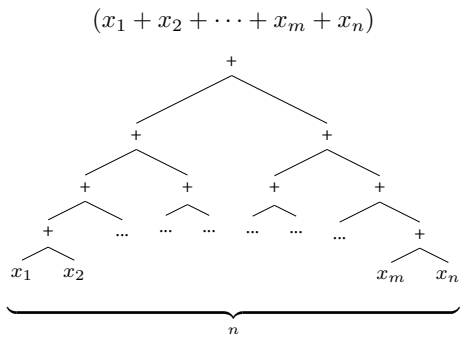


Figure 5.5: This figure illustrates the form of the computation tree that is used to perform parallel reductions in Fcc. Note that CUDA provides no support for nested parallelism, and so, the computation trees must be encoded in the form of one-dimensional arrays, where the children of the node at index i are at indices $2i + 1$ and $2i + 2$, and the parent of the same node is at index $(i - 1)/2$. The workload of a single computation tree can then be parallelised by allotting an equal number of leaf nodes to each thread.

as illustrated in Figure 5.5[8]. This causes a lowering in the number of serial steps that we need to take to compute the sum or the product of a list of numbers, namely from $O(n)$ to $O(\log n)$ for a list of n elements [BGS94].

5.12 Code Generation

The final phase of the compilation process consists of two sub-processes, namely the process of generating a C/C++ program which represents the optimised derivation of the input program, and the process of compiling this program into an executable using the external build tools. The component which is responsible for the generation of the C/C++ program consists of two different code emitters, one for generating standard C99 code and one for generating CUDA C/C++ code. As mentioned in §4.4.5 on page 36, the code generation process is essentially an in-order, depth-first tree traversal where each code emitter outputs appropriate code blocks upon every node visit. We will now look in detail at the implementation-specific details of each of the two code emitters that have been implemented for Fcc.

5.12.1 Sequential C Code Generation

The code emitter which generates standard C99 code is capable of targeting both 32-bit and 64-bit host architectures. All data entities, except from lists, are declared using the primitive typing constructs of the C language. With regard to lists, there are two evident representations available to us: either we can use a linked-list representation or we can choose a flat memory model which implements lists in the form of arrays. In Fcc, we opt for the latter representation as this simplifies the transferring and handling of data

[8]Note that the out-of-order computation that gets performed during the parallel reduction of a list works because of the guaranteed associativity of the reduction operation. Other reduction operations, such as `min` and `max`, can also be implemented as they are both associative operations.

that get performed by all parallelised sections of the program when we are targeting CUDA-enabled devices.

Even though lists are immutable constructs in Fcc, we settle upon a mutable list representation which allows for in-place editing and updating of lists – this is an unproblematic decision as the preprocessing of the language will ensure non-violation of the immutability property of lists. The compiler exploits the mutability of the list representation to reduce the memory footprint and the overhead that is caused by extraneous memory allocations, deallocations and transfers. As already mentioned, the flat memory model uses arrays to represent lists. The elements of each of these arrays are given a type identical to the base type of the parent list, *i.e.*, the type integer or real.

Time and space efficiency of list operations, such as prefixation, can be seen as a requirement of functional-language compilers, since functional-style programs tend to demand high dynamicity of lists. To meet the demands of functional-style programs, the list representation that is employed in Fcc has been made lightweight and efficient. Each list is trivially represented by a pointer to the first element of the list, the length and the capacity. All lists get allocated memory buffers in a block-wise fashion, and as soon as the space that is required to store the elements exceeds the capacity of the allocated memory buffer, the buffer gets resized. When a list overruns its capacity, this normally implies that the size of the allocated memory buffer gets doubled.

As a result of the employed list representation, prefixation is simply a matter of adjusting the pointer to the first element of the list and incrementing the length-field. List prefixation is therefore a fast operation. Due to the need for buffer reallocation when a list buffer gets exhausted, boundary cases can be substantially more costly. However, in practise, these cases are still relatively cheap.

All the special-purpose functions that are described in §A.3 on page 85, map directly onto C library functions, such as the ones that are declared in "`math.h`", and statically defined Fcc functions, which deals with list operations and so on. The only exception is the `if`-function, which gets translated into an `if`-statement. It is worth noting that the resulting `if`-statement does not preserve the expected laziness of the evaluation of the true- and false-branches. If lazy evaluation is desired, the user should use pattern matching or function conditionals instead.

Fcc is highly customisable and provides a huge set of compiler options. These options allow the user to filter out certain stages of the compilation process, and can, in the case of sequential C code generation, also be used to control the GCC maker object. The latter fact implies that the user can specify the optimisation-, debug- and verboseness-flags of the underlying GCC compiler through the Fcc interface.

5.12.2 CUDA C/C++ Code Generation

The CUDA C/C++ code generator is an extension of the sequential code generator that was described in §5.12.1 and provides the following additional functionality:

- Parallelisation of Mapping Constructs – The CUDA code generator identifies all looping constructs that have been tagged for parallelisation, and generates both a CUDA kernel and a block of sequential C code for each loop. More specifically, all mapping constructs are turned into run-time checks with two conditional branches, one for sequential execution and one for parallel execution. The checks reflect the performance model that was derived in §5.11 on page 55. Thus, the generated GPU kernels will only get invoked if there is at least one CUDA-enabled

device available on the host system, and if the considered loops are deemed to run faster in parallel than they are sequentially on the CPU.

The execution of code blocks that have been tailored for parallel execution involves explicit memory handling for all buffers residing on the GPU. Additionally, parallelisation implies that data must be transferred between the host and the device to provide arguments to the GPU kernels and to retrieve the results of the corresponding computations. Furthermore, the number of blocks and threads to use for execution is determined from the size of the processed lists and, thus, code must be emitted to deduce these numbers at run-time. Due to limitations on the number of GPU threads that can be spawned in one go, the code generator also emits code for dealing with cases where a single kernel invocation fails to handle all the elements of the processed list.

- Parallelisation of Reduction-Constructs – As mentioned in §5.10.3 on page 55, idioms such as reduction are matched against optimised templates. More specifically, Fcc generates reduction kernels based on Harris' algorithm for parallel reduction on CUDA [Har08]. The generated reduction kernels can handle both integers and floating-point numbers, and can be used on both 32-bit and 64-bit host architectures.

  The details concerning kernel invocations and memory management are the same for both mapping and reduction constructs. For a full overview, the implementation of the code generator and the methods that are responsible for emitting code for kernel invocations can be found in "`backend/cuda/__init__.py`" and "`backend/cuda/lib.py`," respectively.

- Timing of Looping Constructs – The code generator can also dump code that automatically times the execution of GPU kernels, and these timings can be made to either include or exclude the consequent memory transfers. Similarly, code to time sequential code blocks can be emitted.

- Dumping of Device Information – If the `print_device_info`-flag is set, the code generator emits preludial code that dumps information about all the CUDA-enabled devices that are available on the host system upon entry of the generated program.

Note that the CUDA C/C++ maker object can be used similarly to the GCC maker object to control the underlying CUDA compiler through the Fcc interface.

CHAPTER 6

System Testing and Results

Contents

6.1 Introduction

In Chapter 3, we emphasised the importance of correctness and verifiability. By correctness we mean that all compiled programs must produce the correct answer regardless of what compiler transformations are applied to the intermediate representation of the input. Verifiability is then a matter of being able to confirm that this is the case by extensive correctness testing. The latter also involves performance testing, *i.e.*, testing of what performance gain we can achieve from the employment of the optimisation and parallelisation techniques that have been discussed in previous chapters.

This chapter details the system testing that has been performed to validate the outcome of the project. Furthermore, the chapter also presents the results that have been achieved by the implemented optimisation and parallelisation techniques.

6.2 Unit Testing

After the design of the class hierarchy of the compiler system had been established, a set of unit tests was sketched out – see "`unittests.py`." The following bullet points summarise the targets of the implemented unit tests and provide a manifestation of their importance to the Fcc compiler:

- Types – The type inference system heavily relies on a solid and well-defined implementation of the underlying type system of the Fcc language. Thus, the testing of the type representations has constituted an essential part of the testing process. More specifically, the unit tests that are targeting the type system of Fcc assess

the correctness of the implemented type structure, the type equality checks, the operations that are dealing with type comparisons, the implemented functionality for deducing type dominance, and the various helper functions that are used to derive, for instance, scalar and vector types.

- Abstract Syntax Tree – The abstract syntax tree forms the basis for all the implemented compiler transformations and is a crucial component to target in the testing process. Some of the functionality that has been tested by the unit tests are: node instantiation, invocation of the `update`-method upon node changes, node and branch substitution, type deduction and typification, node collection, and branch duplication.

- Symbol Table – The two components that are listed above constitute a large part of the intermediate representation. However, without an adequate symbol representation, there is little we can do in terms of compilation, optimisation and parallelisation. Thus, the symbol table and the structures that are used to represent symbols and atoms have also been thoroughly tested.

Throughout the life cycle of the project, the main role of the unit tests has been to ensure that the complex representation of the abstract syntax tree conforms to its expected behaviour. Thus, the unit tests have been extensively used in the verification stages of the development process, and have proven to be a particularly useful tool, especially when introducing new functionality to the intermediate code representation.

Considering the fact that the intermediate representation forms the foundation upon which all of the optimisation and parallelisation techniques are performed, the verification of the correctness of this component has been an important part of the testing process. Errors in the intermediate representation are undoubtedly going to propagate to almost all other components of the system. So, it is crucial to eliminate all potential problems in this section of the implementation. The final version of the prototype compiler passed all the implemented unit tests, without exception.

6.3 Black-Box Testing

In Chapter 3, we stated the importance of verifiability, and also mentioned the need for a code base of example programs to support for objective testing of the compiler. During the development of the compiler, we have produced a code base of twenty-odd programs that have been tailored to demonstrate and test the implemented compiler transformations. These test cases can be found in the `test`-folder under the Fcc source code directory on the enclosed CD.

As a supplement to the previously mentioned tests, additional, systematic black-box testing has been conducted. Note that, given the constraints that have been placed upon the project, it has been unrealistic to expect to be able to apply detailed and sufficient testing across all parts of the product. Hence, choices have been made to maximise the benefit of the testing that has been performed, and to minimise the required effort.

The unit tests were carried out to verify the correctness of the underlying structures and components of the system. In contrast, the black-box testing has taken a higher-level approach by testing the general correlation between input and expected output. To sum up, the following transformations have been subject to black-box testing:

- Type Inference – The type deduction phase is a sensitive and crucial stage of the compilation process. Therefore, it has been important to tailor good test cases

to verify the type inference component of the compiler. In addition to having performed exclusive executions of the type inference phase on all the example programs in the provided code base, special user scenarios have been established and tested. All the tests were planned after the type model for the language (§C.1 on page 97) was fixed, but before the implementation phase started, to ensure objectivity.

- Constant Folding and Algebraic Simplification – Methodical tests have been run to ensure correctness of constant folding and algebraic simplification. Boundary and special cases have been tested, *e.g.*, division by zero, multiplication by zero (yielding zero regardless of variable use), *etc.* It should be noted that arithmetic overflows are not picked up, neither by the optimiser at compile-time nor by the generated programs at run-time. This was expected behaviour, and other than that, all the tests were successfully conducted.

- Copy Propagation – Due to the non-destructiveness of assignments in Fcc, one would think that copy propagation is a straightforward transformation. However, the intermediate representation must provide internal constructs for looping, and since loops are iterative, an intermediate representation for destructive assignments is needed to allow the compiler to operate on the bodies of these loops. This complicates the process of copy propagation.

  The testing of this transformation was successful. However, it should be noted that Fcc currently takes a pessimistic approach to copy propagation, *i.e.*, an approach where it does not try to propagate into and past intermediate looping constructs. In further work, the compiler should be made to ignore such cases as long as the considered variable does not appear in the loop body or in the loop conditional.

- Common Subexpression Elimination – This transformation is facing many of the same issues as the copy propagation transformation. In general, intermediate looping constructs have the potential to drastically reduce the benefit of employing common subexpression elimination. However, in practise, such intermediate constructs make infrequent appearances, and rarely impose any real constraints on the compilation process.

  All the test cases for the common subexpression elimination transformation have been tailored to verify that a pessimistic approach is always taken, meaning that candidates are disregarded if there is any chance that a looping construct may interrupt the flow between the two instructions that constitute a candidate. The transformation that deals with collection of equivalent function invocations has been tested in a similar manner.

  Both common subexpression elimination and equivalent invocation collection satisfy the predetermined test criteria. Only a few cases of interrupting looping constructs were identified, and all of these were handled as expected.

- Dead Variable and Dead Code Elimination – In fact, all of the commonly applied compiler transformations (§2.4.2 on page 19) potentially suffer from the issues that have been described above, but again, these issues seldom manifest themselves in practise as loops tend to comprise entire function bodies. Only when multiple recursive functions are applied consecutively in a single function body, do the issues arise, and empirical tests show that there tend to be little correlation between the separated code blocks in these cases.

The black-box testing of the dead variable and dead code elimination transformations were conducted in the same way as with the copy propagation and common subexpression elimination transformations. Additionally, tests were carried out to ensure that unused variables and functions are removed from the intermediate representation as soon as they are rendered as such, and that they, as a result of this, do not get emitted to the generated C/C++ files.

All the tests were successfully conducted, except from in some of the test cases where pattern matching got applied to list parameters. In these cases, two intermediate variables got constructed to represent the head and the tail of the list argument, and these both got flagged as untouchable to the optimiser. Consequently, when further optimisations, or more specifically parallelisation, got carried out, the tail variable ended up being rendered unused. However, since the tail variable was flagged as untouchable, it could not be omitted from the final output, which resulted in the emittance of a redundant variable. Fortunately, this kind of undesired behaviour does not affect the correctness of the program. Also, it should be noted that such instances of unused variables get filtered out by the GCC and CUDA compilers.

- Tail-Call Elimination – The black-box testing of the tail-call elimination transformation entailed multiple crucial points. First of all, tail-call elimination generates either a `while`-loop or a `for`-loop, depending on what information is deducible from the recursive function, *i.e.*, induction variables, *etc.* Note that this transformation only applies to recursive tail calls and, thus, that a successful elimination always will yield an iterative construct. Explicit testing has been performed to ensure that `for`-loops are generated whenever a single, well-defined induction variable can be deduced.

  As opposed to what is the case in the optimisation of augmenting recursion, tail-recursion optimisation can handle recursive functions with multiple base cases. Functions with multiple recursive calls are not dealt with by the compiler, and are therefore ignored.

  A critical point in tail-recursion optimisation is the updating of the induction variables. For instance, if one of the induction variables gets updated before one of the other variables do, and if the other variable is dependent on the original value of the first variable, then we might end up with an invalid state of the program unless temporaries are used to retain the pre-update values of the variables and these retained copies are used in the update process itself – for an example, see §5.4 on page 53. Due to the criticality of this use case, it has been explicitly targeted in the testing process.

  The black-box testing of the tail-call elimination transformation was successful, and revealed no issues.

- Optimisation of Augmenting Recursion – Rigorous testing of the optimisation of augmenting recursion has perhaps been the most important part of the black-box testing, and has therefore received the most attention in the testing process. To verify the functionality of this compiler transformation, numerous tests, targeting the following cases, were constructed: validation of recursive functions with one base case and one recursive call, invalidation of functions with multiple base cases, invalidation of functions with multiple recursive calls, invalidation of functions which implement simultaneous or mutual recursion. Also, multiple inductive step sizes were tested, and the expected results were confirmed.

- Procedure Inlining – This transformation was employed in the Fcc compiler with the intention to simplify the semantic analysis of the input program. Procedure inlining has the potential to reveal otherwise hidden optimisation candidates and, hence, expands the number of optimisation opportunities that are available to us in the input program.

  There are several potentially precarious stages in the inlining process. If not properly implemented, we can run into situations of, for instance, variable duplication, which breaks with the non-destructive assignment strategy. To validate the workings of the inlining transformation, we have made sure that the test cases exploit situations where multiple invocations to the same function appear in the same function body. We have also tested the candidate selection function and verified that multi-clause, non-trivial and non-leaf functions are all being invalidated.

- Loop Fusion and Restructuring – Since only a few of the discussed loop transformations have been implemented, the extent of the black-box testing for loop transformations has been confined to cover loop fusion and loop restructuring. Loop fusion only transpires in cases where the compiler can identify two or more consecutive loops that are working on the same data in a single code block. And loop restructuring is solely concerned with the conversion of generated `while`-loops into equivalent `for`-loops based on the iteration variables that can be identified in the original loop. The latter transformation is in truth a step in the optimisation of recursive functions, and is tested indirectly by the test cases that are targeting the augmenting recursion transformation.

  The loop fusion transformation passed all the considered test cases and revealed no unforeseen problems.

- Procedure Cloning – This transformation is merely concerned with the parallelisation stage of the compilation process, or more specifically, the process in which duplicate functions for parallel execution are created. As part of the black-box testing of the procedure cloning transformation, a manual inspection of the generated CUDA files was carried out for all the programs in the provided code base to ensure that valid GPU kernels were generated. All tests were conducted successfully, except from in some cases where additional arguments needed to be passed into the generated kernels.

The lexical and syntactic analysers have also been subject to black-box testing. Exhaustive test cases were tailored to verify the correctness of the Fcc grammar and its actual implementation. The conducted test cases were written specifically to target known boundary and special cases, and were all passed.

Furthermore, the application of the performance cost model has been tested and verified. However, these tests did not go into as much detail as the previous ones. The black-box testing of the performance model was performed on the examples that can be found in the provided code base, but also on a set of handcrafted scenarios that were written to exploit potential problems of the model. The results were analysed by manual inspection of the produced CUDA files, and no problems were identified. However, it is likely that the model can be improved further by gathering and considering more empirical data that is concerned with the run-time behaviour of certain programming constructs, both on sequential and parallel hardware architectures.

The output from the automatic parallelisation stage, *i.e.*, from transformations such as loop idiom recognition, *etc.*, has been manually inspected and validated for a range of test cases. So have also the results from the C99 code generation.

Lastly, the provision of feedback has been tested by manually verifying that all the expected error and warning messages get produced and presented to the user upon bad input. More specifically, correct provision of the messages that are listed in §5.4 on page 43 has been tested for, and all the tests passed successfully. However, in some cases the column number of the token that was associated with the reported error or warning message did not correspond well with the source of the reported problem and, thus, did not provide any sensible information to the user. Consequently, Fcc has been altered to only report line numbers to avoid confusion.

6.4 Results

As we have seen in the previous sections of this chapter, we have performed both unit testing and black-box testing to ensure correctness of the implemented solution. The unit testing was carried out to validate the building blocks of the final product, and the black-box testing was conducted to assess the soundness of the implemented optimisation techniques. In addition to the unit and black-box testing, performance testing has been conducted to provide metrics which evaluate the proficiency of the implemented compiler transformations.

Due to various limitations of the Fcc language, we have been unable to employ any publicly available benchmarking suites to measure the performance that can be gained from applying the implemented transformations. The limitations include, for instance, the inability to deal with multi-dimensional and heterogeneous data.

Since no public benchmarking suite can be utilised, we have been required to implement a set of test cases ourselves. Considering that this is early work, the limitations of the language has not had a direct influence on the project, as we have been more interested in exploring what can be achieved by the employed compiler transformations than the possibilities that are presented to us by the language. This argument has led us to opt for a minimalistic language which is easy to analyse and process from a semantic perspective. Having said that, given the aim of providing programmers with a tool that simplifies development of parallel algorithms, the language should surely be extended as part of further work.

Due to the simplicity of the current revision of the Fcc language, parallelism only manifests itself in a limited number of ways, namely in the form of primitive mapping and reduction. The main objective of the performance testing is to explore how these manifestations affect the running time of the generated programs.

It should be noted that, by testing the performance of programs that have been parallelised by Fcc, the performance cost model will be tested as well. However, the cost model has already been tested, as part of the black-box testing that was presented in §6.3, and is therefore completely left out of the discussion of this section. Throughout the project, the main focus has been put into what sequential constructs can be parallelised for data-parallel architectures, and the tests that have been carried out as part of the performance testing reflect this focus. Furthermore, this means that the details around the selection of parallelisation candidates are omitted from this section.

The test cases that have been considered in the performance testing are concise and objective. They have been written to attack the core of the parallelisation process to help identify what speedups can be achieved by the employed transformations. Obviously, only attainable parallelisation constructs have been targeted.

We have constructed and tested six different parallelisation scenarios, namely the ones listed below. Observe that both mapping and reduction constructs have been ac-

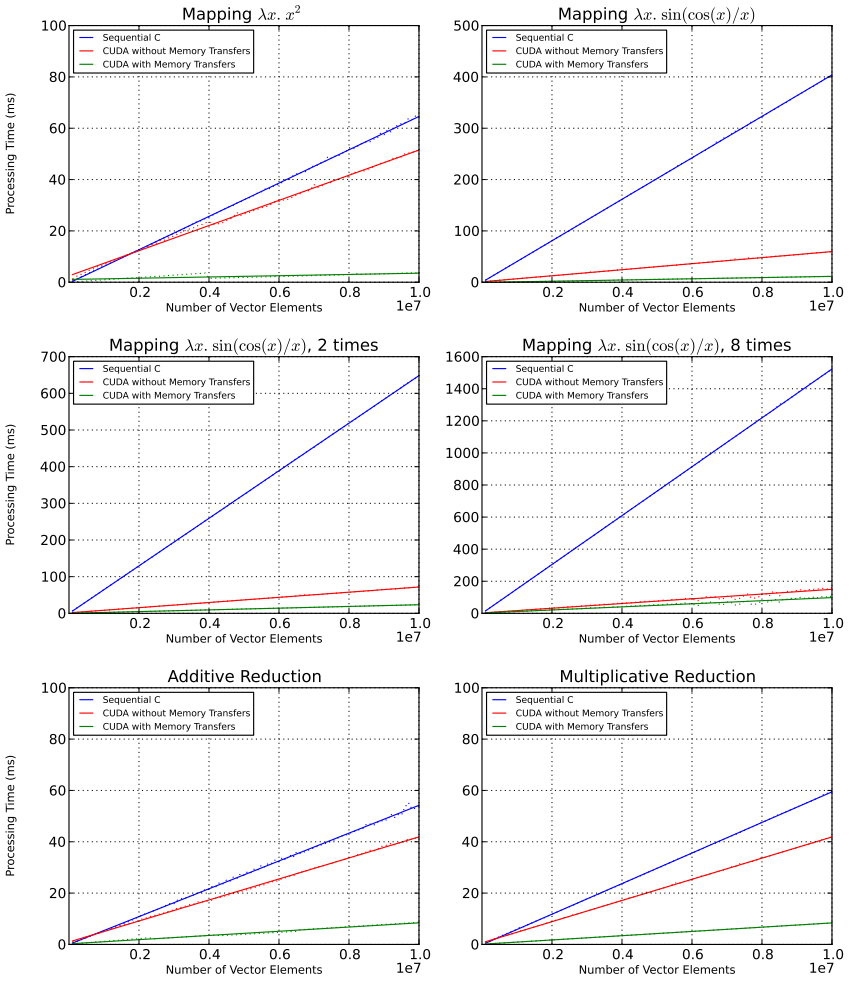


Figure 6.1: This figure presents the results of the performance testing that was carried out on a system with the following specifications: NVIDIA GeForce GT 330M (512 MB), Apple Mac OS X Snow Leopard 10.6.7, Intel Core i7 2.66 GHz, 64-bit system, 32-bit executable, 8.00 GB RAM.

counted for. Also note that additional mapping constructs with kernels of different computational complexity have been tested. This is to explore how the execution time of a GPU kernel affects the overall performance of a parallelised looping construct, and how this effect compares to the performance of an equivalent sequential loop.

(a) Mapping of all the elements of a list, yielding the square of each element.

(b) Mapping of all the elements of a list, yielding $\sin(\cos(x)/x)$ of each element, x.

(c) The same as (b), but applied twice.

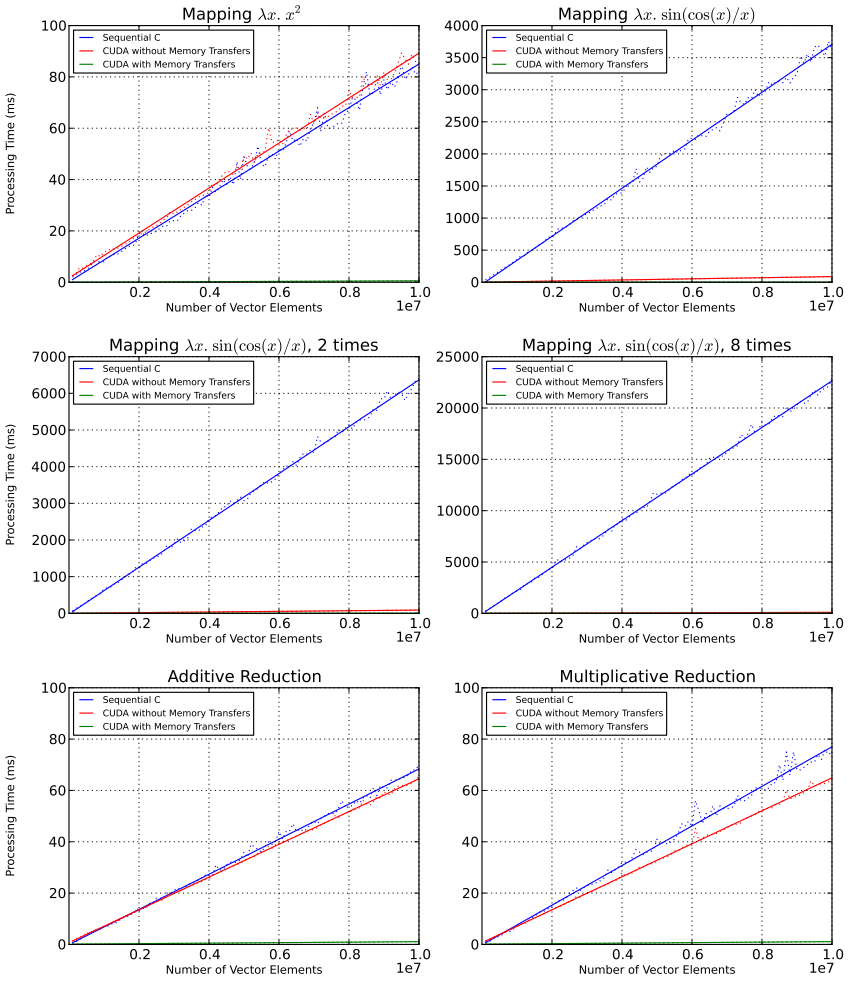


Figure 6.2: This figure presents the results of the performance testing that was carried out on a system with the following specifications: NVIDIA GeForce GTX 480 (1536 MB), Microsoft Windows Server 2008 R2 Enterprise, Intel Core 2 Duo CPU E6850 3.00 GHz, 64-bit system, 32-bit executable, 4.00 GB RAM.

(d) The same as (b), but applied eight times.

(e) Additive reduction of all the elements of a list.

(f) Multiplicative reduction of all the elements of a list.

Generally, the results that have been achieved are good in terms of performance, especially considering the fact that executables that are produced from functional-style code often tend to be slower than similar implementations in imperative languages because of the high-level of abstraction that is common in functional-style code, *e.g.*, because of

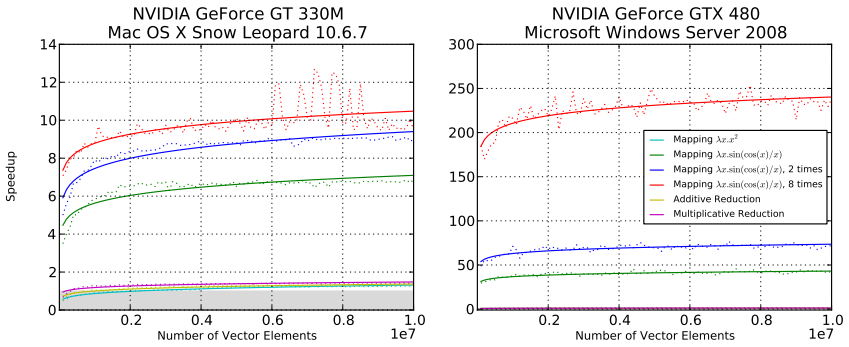


Figure 6.3: This figure presents the speedup results from the conducted performance tests. We observe that the maximum achieved speedups on the two test machines were 12.6 and 253.3 times the sequential execution speeds, respectively. Generally, we achieved a tenfold speedup on the OS X machine and a hundredfold speedup on the Windows machine. We observe that the vast difference in speedup is mainly caused by the difference in sequential execution speed on the two machines, and that the parallel execution speed was fairly consistent between the two environments.

recursion, higher-order functions, garbage collection, automatic storage management, *etc.* The application of general optimisation techniques to sequential code has yielded speedups of up to 20 times the original execution speed for small and trivial example programs. These impressive results are primarily an outcome of the elimination of recursion which demonstrates the value of the optimisation of augmenting recursion. Higher speedups are expected for programs that are larger and more complex than the ones that have been subject to testing – the ones in the provided code base.

Example of Sequential Speedup: The sequential optimisation of a program computing the value of the factorial function, Listing 5.1, applied to $N = 40$, *i.e.*, `fact`(40), resulted in an average speedup of 16.4 when the optimised program was compared to a bare, unoptimised version.

In terms of parallelisation, the initial results have been promising, see Figure 6.1 and Figure 6.2[1]. However, we observe that to achieve good speedups, we are often required to deal with fairly costly, or at least non-atomic, GPU kernels. For instance, scenario (a) results in little, if any, speedup, as the overhead that is inflicted by the memory transfers between the host and the device cancels out the performance that is gained from executing the kernel in parallel. The observation is also corroborated by the fact that scenarios (b), (c) and (d) achieve speedups that increase with the complexity of the kernels. In terms of parallel reduction, we observe that there is little difference between additive and multiplicative reductions, and that the achieved speedups are low, especially if we include the overhead that is caused by memory transfers between the host and the device. However, in more complex applications, some of these memory transfers can

[1]Note that the sequential reference points of the graphs that are presented in this chapter are measures from the execution of fully optimised, sequential programs, *i.e.*, the speedup is a measure of the form t_s/t_p, where t_s is the running time of the fastest sequential representation of the problem, and t_p is the running time of the fastest parallel representation of the problem.

possibly be made redundant due to memory reuse, and the actual speedup may then be higher. Observe that the graphs in Figure 6.3 show speedups where these memory transfers are included in the calculation, and that Figures 6.1 and 6.2 show clear benefits from parallelisation if we disregard the cost of the memory transfers. Obviously, we cannot disregard the transfer costs altogether, but a slight reduction in the number of transfers compared to the number of actual kernel invocations can contribute significantly towards faster execution speeds. Also, it should be noted that the merger of mapping and reduction constructs will cause higher speedups, and that the identification and transformation of consecutive parallelisation candidates should be extended and fine-tuned in further work.

We observe that the distinct differences in the speedup figures for the two test machines are mainly caused by the slower execution speed of sequential programs on the Windows machine. The speed of execution of CUDA kernels, including memory transfers, has proven to be relatively consistent across the test environments, whilst there has been a vast variation in the speed of sequential execution. This illustrates the advantages of parallelisation, especially on slower machines.

To obtain more compelling results, we would have to extend the language to allow for more complex and interesting problems to be computed, *e.g.*, support for multi-dimensional data structures and complex number types would enable us to compute 2D Fourier transforms, and other commonly parallelised algorithms. This would also evoke work into how these multi-dimensional data structures can be represented in memory to optimise the processing performance on graphics processors, and into how the memory layout in general affects the delicate memory models of such devices – see §2.2.2.

Finally, it is worth noting that all the tests have been conducted on large data-sets, spanning in size from 100 000 to 10 000 000 elements, and that to obtain objective time measurements, multiple, subsequent kernel executions have been performed for each of the test cases. The final analysis has been based on the average running times of ten test runs. Generally, single-instance execution of GPU kernels can be slow due to the need for initialisation of the communication channel between the host and the device. Therefore, the GPU kernel should always be executed prior to the first time measurement to ensure reliable results [SK10].

CHAPTER 7

Conclusion

Contents

7.1 Summary

In this report, we have seen how the renewed urge for parallelism over the past few years has resulted in new and innovative solutions to parallelisation. The way programmers have started to utilise graphics processors in general-purpose computing is an example of this. We have also touched upon the difficulty of parallelising existing, sequential algorithms and, thus, discussed some of the challenges that are present in parallel computing today.

Chapter 1 started out by presenting the background and motivation of the project before listing the overall aims and objectives of the work. We recap the objectives by recalling that our main aims were to design a statically and implicitly typed, functional programming language and to implement a prototype compiler for this language to try to exploit the massively data-parallel processing capabilities of CUDA-enabled devices.

In the literature review in Chapter 2, we stated that the implemented compiler should aim to simplify the development of algorithms for data-parallel architectures, and, again, that it should endeavor to produce fast, parallel code. However, we also concluded that the development of the compiler was to focus on the preparation phase of the parallelisation process, *i.e.*, on the transformations that can be employed to prepare seemingly sequential, functional-style code for parallelisation onto graphics processors.

The literature review further examined the technical details around CUDA, and explored the world of parallel software libraries and programming languages to gather insight into how these map onto data-parallel problems. The chapter also looked at NESL

and how it implements a language-based performance model to reason about the time and space complexities of parallel implementations. Furthermore, the chapter discussed various aspects of the static code analysis that has been employed in parallelising versions of Fortran, and collected ideas related to code readability and analytic amenability from various concurrency-oriented and functional languages.

Lastly, the chapter argued how automatic parallelisation of functional-style code is a matter of converting recursive functions into parallelisable loops, and furthermore, discussed the details around data-dependence and data-flow analyses, and the different transformations that can be applied to the input to achieve the above objectives.

Chapter 3 presented the functional and non-functional requirements of the implemented solution, and observed the importance of correctness, verifiability, feedback and actual performance gain. The requirements specification also listed the limitations of the system and some of the functionality that has been left out due to lack of practical relevance. Lastly, the chapter disclosed the testing requirements and all the resources that have been used in the performance testing of the compiler.

The report then moved on to discussing the high-level design of the implemented solution in Chapter 4. The tools and modules that have been used in the development process were presented, and a brief description of the Fcc language was given. Furthermore, the language design was justified. Chapter 4 then concluded by giving a presentation of the high-level software architecture of the product, to introduce the reader to the overall compilation process.

Chapter 5 entailed a thorough review of the low-level design of the system, and aimed to provide the reader with a comprehensive overview of all the implementation-specific details of the compiler. The chapter illustrated and explained the workings of the implemented solution, and showed the importance of a well-behaved and rigorously tested intermediate representation. Type deduction was described and demonstrated, and so were also the commonly applied compiler transformations. In addition, the chapter outlined the delicacy of some of the implemented transformations.

Next, a detailed presentation of the interprocedural transformations was given, and the importance of the conversion of recursion into iteration was emphasised[1]. Furthermore, loop transformations and the transformations that are involved in the process of automatically parallelising sequential code were presented. The chapter culminated in a presentation and discussion of the implemented performance model, before explaining the workings of the code generator in detail.

Chapter 6 outlined the test plan and gave a report of the test results. The momentousness of unit testing was highlighted and, further, justified by showing how errors on a component level easily propagate to other parts of the system. More specifically, the intermediate code representation was identified as a crucial test target. The unit testing section concluded that all the test cases were successfully conducted.

The testing chapter also argued the significance of black-box testing, and accounted for how these tests allow us to reveal potential problems of the implemented solution. The testing was conducted successfully. However, three minor issues were identified. These are summarised in §7.2.

The last section of Chapter 6 presented the results of the performance testing. Except from finding that the current employment of optimisation and parallelisation techniques has yielded promising increases in execution speed, a few remarks were made with respect to further work, see §7.2. In particular, it was concluded that various limit-

[1]It should be noted that the interprocedural transformations are described in more detail in Appendix B, together with justifications and proofs for the correctness of some of the applied optimisation techniques.

ations of the current language design and the current compiler implementation disallow parallelisation of more interesting and complex programming constructs. For instance, computation over multi-dimensional data-sets is not supported, and instances of simultaneous recursion are not parallelised (§7.4 on page 74). Nevertheless, tenfold to hundredfold speedups have been achieved in the conducted test cases, depending on the point of comparison, and these are satisfactory results, especially considering that the Fcc language and the presented compiler are currently in early stages of the development and have a long way to go to reach their full potential.

7.2 Results

This section briefly summarises the results of the black-box and performance testing. The black-box testing and its findings were presented in §6.3 on page 61, and the results of the performance testing in §6.4 on page 65.

7.2.1 Black-Box Testing

The results of the conducted black-box testing can be summarised as follows:

- The pessimistic approach of the commonly applied compiler transformations should be subjected in further work to allow for a more global application of these transformations. As of now, intermediate looping constructs will interrupt the flow of the dependence analysis and break the considered basic blocks into several smaller blocks, even if the considered variables do not appear in the looping constructs themselves.
- In rare cases, variables do not get pruned when they have been flagged as unused. The reason for this is that they in these cases, intentionally, also get flagged as untouchable by the code analyser to ensure that the optimiser does not alter any of the containing statements. This does not really constitute a problem as the lack of pruning is compensated for by the external build systems.
- The column numbers of the tokens upon which some of the error and warning messages are reported, are sometimes misleading as they do not refer back to the actual root of the problem. Consequently, the current version of the prototype compiler only reports line numbers.

Apart from the issues that have been listed above, we discovered no problems in this stage of the testing process. In short, the black-box testing was considered successful. Note however that, whilst the black-box testing has been conducted successfully, there has also been identified a number of areas of improvement to the Fcc language and the code optimiser. These are discussed in §7.4 on page 74.

7.2.2 Performance Testing

The observations and results of the conducted performance testing can be summarised as follows:

- No public benchmarking kit has been used in the performance testing due to various limitations of the Fcc language, *e.g.*, the lack of support for multi-dimensional data structures. Instead, a code base of example programs has been used. It is

generally noted that the language should be extended in further work to allow for more complex problems to be computed and tested.

- Parallelism only manifests itself in a limited number of ways, more specifically, as mapping and reduction constructs. Consequently, the input programs must be on a naturally data-parallel form, where mapping and reduction can be exploited, to benefit from the employed compiler transformations.

- The focus has mainly been put into the exploration of what sequential constructs can be parallelised for data-parallel execution. Also, as the work has focused on the preparatory transformations for parallelisation, the final stage of the parallelisation process has received little attention. The Fcc compiler generates mapping and reduction kernels. However, in further work, the merging of consecutive kernels should be extended to allow for merging of heterogeneous collections of kernels as well.

- Six scenarios have been tested to explicitly test what performance can be gained from applying the employed parallelisation transformations. The following observations have been made:

  - Generally, good results have been achieved. More specifically, the parallelisation of the six test scenarios yielded tenfold to hundredfold speedups. The achieved speedups may seem fairly low compared to what is normal for GPGPU programming, at least when we take consistency into account. However, given the current limitations of the Fcc language, and how these restrict the expressiveness of the programmer, these results actually exceed the expectations prior to commencement.
  - Optimisation of simultaneous recursion could allow for parallelisation of operations that span across multiple vectors, and such constructs are expected to achieve higher speedups. A natural move in further work is therefore to consider optimisation techniques that deal with simultaneous recursion.
  - The effect of the automatic parallelisation is dependent on the computational cost of the generated CUDA kernels. As noted in §6.4 on page 65, the attainable speedups increase steadily with the complexity of the kernels.
  - Parallelisation is particularly beneficial when targeting slower machines. As we observed in §6.4, the test machine with the lowest sequential execution speeds, which is still a reasonably fast and modern machine, achieved up to 250 times speedup from parallelisation, whilst the fastest machine only achieved a shy tenfold increase in execution speed.
  - The tests were conducted on large data-sets. Not only did this demonstrate the benefit of parallelisation in data-intensive processing, but it also helped to ensure objectivity as the shorter the execution time is, the more likely it is that the results are going to be influenced and spoiled by the scheduling mechanisms of the operating system. Normally, by considering large data-sets, the inflicted system overhead, with respect to time, are reduced in comparison to the time it takes to compute the answer and, thus, will have less impact on the performed time measurements.

- In the context of sequential programming, it should be noted that up to 20 times speedups have been achieved in the execution of small and trivial, sequential pro-

grams. These speedups are predominantly a result of the application of interprocedural transformations, such as, conversion of augmenting recursion into tail recursion and tail-call elimination.

7.3 Critical Evaluation

The goals of designing a functional programming language and implementing a compiler for the designed language have been reached. Even though the implemented solution is regarded as a prototype compiler, the work has proven to attain satisfactory optimisation results, even from having to deal with a limited and minimalistic language.

All the functional and non-functional requirements that were specified in Chapter 3 on page 28 have been satisfied. However, the goal of making a tool that simplifies the development of parallel algorithms has not been fully reached. The syntax and grammar of Fcc do indeed allow for easy expression of functional-style problems, but the language only supports a limited number of types and, more crucially, only supports processing of one-dimensional data. This reduces the applicability of the language to complex computing problems. Saying that, we did not expect to fulfill the desire to make a tool that simplifies the development of parallel algorithms in the scope of this project, as this has been established as a long-term goal.

7.4 Further Work

This section presents ideas for further improvements of the implemented compiler system and proposes concepts for further work.

- Extended Type System – In the current implementation, only integers, reals, and one-dimensional, homogeneous data structures are supported. However, as we have seen multiple times in this report, the available type system imposes undesired limitations on the programmer. Thus, further work should consider adding support for multi-dimensional and heterogeneous data structures, and complex numbers.
- Simultaneous Recursion – As mentioned in §7.2, the optimisation of simultaneous recursion could be employed to allow for parallelisation of operations that span across multiple vectors. For instance, consider the `add`-function in Listing 7.1. The current version of Fcc is unable to parallelise such functions due to various constraints that are placed on the optimisation of augmenting recursion – see Appendix B. Parallelisation of vector operations such as vector addition and vector multiplication tends to yield higher speedups than the parallelisation of operations that are concerned with a single vector, such as maps[2]. Thus, by implementing a transformation that can convert simultaneous recursion into iterative constructs, new potential can be exploited by the compiler. It follows from the proofs in Appendix B that this transformation is routine to implement.

Listing 7.1: Simultaneous Recursion

```
# note that this function only deals with lists of equal lengths
add([],      []):      [];
add([H1|T1], [H2|T2]): [H1+H2|add(T1, T2)].
```

[2]The argument follows from empirical testing of native CUDA programs.

- Multiple Recursive Calls – Instances of augmenting recursion may wind up with more than one recursive call in the last statement of the recursive function clause, *e.g.*, consider the Fibonacci function in Listing 7.2.

Listing 7.2: Fibonacci Function (Plain)

```
fib(0): 1;                  # base case 1
fib(1): 1;                  # base case 2
fib(N): fib(N-1) + fib(N-2). # recursive case, with ...
                            # ... multiple recursive calls
```

Future versions of Fcc should try to capture these cases and convert them into tail-recursive equivalents. For instance, by applying the techniques that have been discussed by Liu and Stoller [LS00], one can derive a tail-recursive Fibonacci function similar to the one that is presented in Listing 7.3.

Listing 7.3: Fibonacci Function (Tail-Recursive Form)

```
fib(0, P1, P2): P1;                 # base case 1
fib(1, P1, P2): P2;                 # base case 2
fib(2, P1, P2): P1 + P2;            # base case 3
fib(N, P1, P2): fib(N-1, P2, P1+P2). # recursive case
fib(N): fib(N, 0, 1).               # auxiliary function
```

- Compute-Intensive Loops – Some instances of non-structural recursion can involve a substantial amount of computation per recursive step. The programmatic representation of such instances can appear in the form of a loop. If these loops yield scalar results, each iteration is only dependent on the results that are computed in the previous iteration, and all inter-iteration computations are carried out with associative operations, we can still exploit the GPU to improve the running time of the loop. This can be done by extending the existing implementation of parallel reduction.

- Recursion Detection – There are many areas of improvement in the system for detection of well-behaved recursion. First of all, the current implementation only deals with multiple base cases if the recursive function implements tail recursion. A function which implement augmenting recursion can only be subject to optimisation if it has a single base case.

  Also, even if tail-recursive functions with multiple base cases get translated into loops, this does not mean that the resulting loops will be parallelised. On the contrary, only tail-recursive functions with a single base case get parallelised in the current version of Fcc.

  Another potential problem is that the collection of equivalent invocations can end up hiding the fact that a function is recursive. For instance, if the last statement of a recursive function clause is $g(N-1) \times g(N-1)$, this will yield an internal representation, $t_1 \leftarrow g(N-1);\ t_1 \times t_1$. As a result, the compiler will be unable to deduce that the function clause is recursive, and the rest of the tests lapse.

  Lastly, support for mutual recursion and arbitrary step sizes should be added – per now, the compiler only deals with step sizes of positive and negative one.

- Preallocation of Lists and Generator Templates – Sometimes, when dealing with recursive functions that generate or manipulate lists, the performance can be improved substantially by preallocating the memory that is used by the destination list. Preallocation allows us to set the values of the list elements by random access during the population / manipulation process, rather than having to run a prefixation operation for each iteration. Such preallocation mechanisms can also be made to detect cases where the destination list need to be, say, double or half the size of the original list.

- Loop Optimisations – By introducing multi-dimensional data structures to the Fcc language, and by improving some of the existing compiler transformations, effective loop optimisations will become more important in the compilation process, as these alterations will imply that nested and more complex loops can get generated from the input. Hence, several of the unimplemented loop transformations that were presented in Chapter 2 should be considered in further work.

- Run-Time Checks, Exception Handling and Garbage Collection – As identified in Chapter 6, Fcc performs no boundary or overflow checks, not at compile-time nor at run-time. Similarly, exceptions like division by zero are never handled.

  Run-time checks that can help the programmer to identify the errors that are mentioned above should be implemented in further work, and the employment of these run-time checks should be made optional, *i.e.*, it should be possible to turn them on and off so that the user can choose whether the generated program should perform the checks or not.

  It should be noted that the compiler does not produce any code for sophisticated garbage collection, and that in the current version of Fcc, the generated programs simply keep track of all allocated memory and deallocate all buffers upon termination. This functionality should be improved in further work.

- Derivation of Operational and Denotational Semantics – To be able to provide more thorough reasoning about Fcc programs, it would be useful to derive the operational semantics for the Fcc language. From a reasoning point of view, it would be ideal if also the denotational semantics could be derived, together with a description of the relationship between the two formal semantics [NN07].

- Continuation-Passing Style – A topic for further work could be to explore the benefits of employing a continuation-passing style intermediate representation, and whether such an employment would exploit more optimisation candidates.

- Task Parallelism and Nested Parallelism – The reduction kernels that get generated by the compiler, compute the appropriate reductions of the input lists, and do this by employing a nesting strategy, see §5.11 on page 55. The reduction operations can in fact be regarded as emulations of nested parallelism, which is a phenomenon which is often employed on task-parallel architectures.

  Recent versions of the CUDA framework allow for multiple pseudo-simultaneous kernel invocations on devices of compute capability 2.0 or higher, and this provides a means to accomplish a limited form of task parallelism. Furthermore, streaming and other techniques can be used to extend the notion of concurrency.

  Further work should explore how these features can be exploited to add a limited form, or at least an illusion, of task parallelism to Fcc. It should be noted that,

since CUDA does not explicitly support task parallelism due to the nature of the targeted devices, there are limitations to how far we can push the illusion. Regardless, it is likely that some of these features can be exploited in the parallelisation process and, thus, be used to improve the performance of the produced output.

Bibliography

[ABC+88] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the 1st International Conference on Supercomputing*, pages 194–211, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[AK81] J. R. Allen and K. Kennedy. *PFC: A program to convert Fortran to parallel form*. Rice University, Department of Mathematical Sciences, 1981.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, August 2006.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[App98a] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[App98b] Andrew W. Appel. SSA is Functional Programming. 1998.

[ARB08] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, May 2008. Retrieved: 8 December 2010. `http://www.openmp.org/mp-documents/spec30.pdf`.

[Arm07] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[ASSP96] H. Abelson, G. J. Sussman, J. Sussman, and A. J. Perlis. *Structure and Interpretation of Computer Programs*, volume 2. MIT Press, Cambridge, MA, USA, 1996.

[AVWW93] J. Armstrong, R. Virding, C. Wikstrom, and W. Williams. *Concurrent programming in ERLANG*. Prentice Hall, 2nd edition, 1993.

[BCKT79] U. Banerjee, Shyh-Ching Chen, D. J. Kuck, and R. A. Towle. Time and Parallel Processor Bounds for Fortran-Like Loops. *Computers, IEEE Transactions on*, C-28(9):660–670, 1979.

[BD77] R. M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *J. ACM*, 24:44–67, January 1977.

[BG95] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 226–237, New York, NY, USA, 1995. ACM.

[BG96] Guy E. Blelloch and John Greiner. A Provable Time and Space Efficient Implementation of NESL. *SIGPLAN Not.*, 31:213–225, June 1996.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26:345–420, December 1994.

[BHC+93] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-Parallel Language. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 102–111, New York, NY, USA, 1993. ACM.

[Bir77] R. S. Bird. Notes on Recursion Elimination. *Commun. ACM*, 20:434–439, June 1977.

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The ParaScope editor: An interactive parallel programming tool. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 540–550, New York, NY, USA, 1989. ACM.

[Ble95] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (3.1), September 1995. Retrieved: 8 December 2010. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-95-170.html`.

[Ble96] Guy E. Blelloch. Programming Parallel Algorithms. *Commun. ACM*, 39:85–97, March 1996.

[Cal10] Peter Calvert. Parallelisation of Java for Graphics Processors. 2010.

[Cha10] M. M. T. Chakravarty. GHC/Data Parallel Haskell, May 2010. Retrieved: 8 December 2010. `http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`.

[CHK92] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure Cloning. In *Computer Languages, 1992., Proceedings of the 1992 International Conference on*, pages 96–105. IEEE, 1992.

[CKL+11] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[CKLP01] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal – Nested Data Parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 524–534, London, UK, 2001. Springer-Verlag.

[DG08] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[For97] High Performance Fortran Forum. High Performance Fortran Language Specification, January 1997. Retrieved: 8 December 2010. `http://hpff.rice.edu/versions/hpf2/hpf-v20.pdf`.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. *SIGPLAN Not.*, 28:237–247, June 1993.

[Har88] L. Harrison. Parcel: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 527–538, New York, NY, USA, 1988. ACM.

[Har05] M. Harris. Mapping Computational Concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, page 50. ACM, 2005.

[Har08] M. Harris. Optimizing Parallel Reduction in CUDA. 2008. Retrieved: 8 December 2010. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[HJ85] Robert H. Halstead Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[Hud89] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, 21:359–411, September 1989.

[ISO97] ISO/IEC 1539-1:1997. *Information technology – Programming languages – Fortran – Part 1: Base language*. ISO/IEC., 1997.

[Jon89] P. Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175, 1989.

[Ken07] A. Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 177–190. ACM, 2007.

[KY04] Dongkeun Kim and Donald Yeung. A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code. *ACM Trans. Comput. Syst.*, 22:326–379, August 2004.

[LH88] James R. Larus and Paul N. Hilfinger. Restructuring Lisp Programs for Concurrent Execution. In *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, PPEALS '88, pages 100–110, New York, NY, USA, 1988. ACM.

[LS00] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 73–82. ACM, 2000.

[Mun10] A. Munshi. OpenCL Specification 1.1. *Khronos OpenCL Working Group*, 2010. Retrieved: 8 December 2010. `http://developer.amd.com/gpu/amdappsdk/assets/opencl-1.1-rev33.pdf`.

[NN07] H. R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer*. Springer Verlag, 2007.

[NO99] S. Nishimura and A. Ohori. Parallel functional programming on recursively defined data via data-parallel recursion. *J. Funct. Program.*, 9(4):427–462, 1999.

[NR98] R.W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[NVI09] NVIDIA Corporation. NVIDIA CUDA Architecture: Introduction & Overview, April 2009. Retrieved: 8 December 2010. `http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf`.

[NVI10a] NVIDIA Corporation. CUDA CUBLAS Library, August 2010. Retrieved: 8 December 2010. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUBLAS_Library.pdf`.

[NVI10b] NVIDIA Corporation. NVIDIA CUDA C Programming Guide 3.1, July 2010. Retrieved: 8 December 2010. `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf`.

[PG10] The Portland Group. CUDA Fortran Programming Guide and Reference, November 2010. Retrieved: 8 December 2010. `http://www.pgroup.com/doc/pgicudaforug.pdf`.

[PGH+89] Constantine D. Polychronopoulos, Milind B. Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *Int. J. High Speed Comput.*, 1:45–72, April 1989.

[Pit05] Andrew Pitts. Typed operational reasoning. In *Advanced Topics in Types and Programming Languages, chapter 7*, pages 245–289. MIT Press, 2005.

[POS95] IEEE Std 1003.1c-1995. *Threads*. IEEE., 1995.

[PW86] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[RRB+08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[SGS95] SGS-THOMSON Microelectronics Limited. occam 2.1 Reference Manual, May 1995. Retrieved: 8 December 2010. `http://www.wotug.org/occam/documentation/oc21refman.pdf`.

[SH92] J. P. Singh and J. L. Hennessy. An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization. *Shared memory multiprocessing*, pages 213–250, 1992.

[Shi05] Olin Shivers. The Anatomy of a Loop: a story of scope and control. *SIGPLAN Not.*, 40:2–14, September 2005.

[SK10] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, July 2010.

[ST98] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Comput. Surv.*, 30:123–169, June 1998.

[Ste77] Guy Lewis Steele, Jr. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 annual conference*, ACM '77, pages 153–162, New York, NY, USA, 1977. ACM.

[Sut05] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[Szy91] Boleslaw K. Szymanski. *Parallel functional languages and compilers*. ACM, New York, NY, USA, 1991.

[TLP02] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *J. Funct. Program.*, 12:469–510, July 2002.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. *SIGOPS Oper. Syst. Rev.*, 40:325–335, October 2006.

[Wol90] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.

[YTT+08] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong. Map-Reduce as a Programming Model for Custom Computing Machines. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, pages 149–159. IEEE, 2008.

APPENDIX A

Language Specification

Contents

A.1 Syntax

Fcc is a statically and implicitly typed functional programming language which has been developed as part of this project. The language implements single assignments and call-by-value evaluation, and is built up from two base types, namely integers and reals. The language can also deal with homogeneous, immutable lists with elements of type integer or real. Furthermore, Fcc supports the use of higher-order functions, meaning that function references can be passed as arguments, stored to variables and returned from functions. The following sections include the formal specification of the Fcc programming language.

A.1.1 Tokens

This section lists the tokens that get identified during the lexical analysis of the compilation process, and their corresponding regular expressions.

atom	:	`[a-z][A-Za-z0-9_]*`	(A.1)
	|	`'[^']*'`	(A.2)
comment	:	`#.*`	(A.3)
dot	:	`[;.]`	(A.4)
empty	:		(A.5)
identifier	:	`[A-Z][A-Za-z0-9_]*`	(A.6)
keyword	:	`and|not|or`	(A.7)
literals	:	`[:=()\[\]$]`	(A.8)
operator	:	`[+*/\\%-]`	(A.9)
	|	`==|!=|>=|<=|>|<`	(A.10)
number	:	`[0-9]+(\.[0-9]+)?([Ee][+-]?[0-9]+)?`	(A.11)
separator	:	`[,]?`	(A.12)
string	:	`"[^"]*"`	(A.13)

A.1.2 Grammar

This section lists the formal, grammatical rules that apply to the language. Note that bold words and symbols indicate keywords and literals, and that the vertical bar is used as an or-operator in the rules (A.41), (A.42), (A.44) and (A.45).

goal	:	definition goal	(A.14)
	|	empty	(A.15)
definition	:	atom **(** param_list **):** stmt_list dot	(A.16)
	|	atom **(** param_list **),** comparison **:** stmt_list dot	(A.17)
stmt_list	:	statement stmt_list	(A.18)
	|	statement	(A.19)
	|	empty	(A.20)
statement	:	identifier **=** comparison separator	(A.21)
	|	comparison separator	(A.22)
elem_list	:	comparison **,** elem_list	(A.23)
	|	comparison	(A.24)
	|	empty	(A.25)
parameter	:	identifier	(A.26)
	|	number	(A.27)
	|	atom	(A.28)
	|	**[ ]**	(A.29)
	|	**[** identifier **|** identifier **]**	(A.30)
param_list	:	parameter **,** param_list	(A.31)
	|	parameter	(A.32)
	|	empty	(A.33)

arg_list	:	comparison **,** arg_list	(A.34)
	|	comparison	(A.35)
	|	empty	(A.36)
function_app	:	identifier **(** arg_list **)**	(A.37)
	|	atom **(** arg_list **)**	(A.38)
comparison	:	comparison **and** comparison	(A.39)
	|	comparison **or** comparison	(A.40)
	|	expression **(==|!=|>=|<=|>|<)** expression	(A.41)
	|	expression **(<|<=)** expression **(<|<=)** expression	(A.42)
	|	expression	(A.43)
expression	:	expression **(+|-|*|\/|%)** expression	(A.44)
	|	**(-|not)** expression	(A.45)
	|	identifier	(A.46)
	|	atom	(A.47)
	|	atom **$** number[i]	(A.48)
	|	number	(A.49)
	|	string	(A.50)
	|	function_app	(A.51)
	|	**(** comparison **)**	(A.52)
	|	**[** elem_list **]**	(A.53)
	|	**[** expression **|** expression **]**	(A.54)

A.2 Operator Precedence

The operator precedence rules for Fcc are defined in Table A.1, and are ordered from lowest to highest. The third column defines the associativities of the operators, which indicate in what order operators of equal precedence are applied.

A.3 Special-Purpose Functions

Fcc provides a set of built-in functions. The following list summarises these:

head(list) Returns the first element of `list`. In other words, if `list` is on the form $[\, l_1, l_2, \ldots, l_n \,]$, **head** returns l_1. If the cardinality of `list` is zero, then the return value is undefined.

tail(list) Returns the list comprising all elements of `list` except from the very first element. In other words, if `list` is on the form $[\, l_1, l_2, \ldots, l_n \,]$, **tail** returns $[\, l_2, \ldots, l_n \,]$. If $n = 0$, then the return value is the empty list.

len(list) Returns the length of `list`.

[i] Atoms on the form `add$2` are treated as function references; in this case to a function of name `add` which has two parameters.

Operator	Description	Associativity
or	Boolean disjunction.	Left-to-Right
and	Boolean conjunction.	Left-to-Right
== != **>= <= > <**	Equality check. Numerical comparison.	Non-Associative Non-Associative
+ -	Addition and subtraction of integers and floating-point numbers.	Left-to-Right
*** / \ %**	Multiplication and division of integers (\) and floating-point numbers (/).	Left-to-Right
not	Logical negation.	Right-to-Left
-	Arithmetic negation.	Right-to-Left
[] [|]	List composition and list prefixation.	Left-to-Right

Table A.1: Precedence and associativity of operators in the Fcc language.

empty(list) Returns a non-zero integer if the length of `list` is greater or equal to one, zero otherwise.

if(cond, tval, fval) Returns `tval` if `cond` is non-zero, `fval` otherwise. Please note that `tval` and `fval` get evaluated eagerly. Fcc is a call-by-value language and does not implement lazy evaluation.

range(from, to, step) Generates a sequence of numbers between `from` and `to` with step size given by `step`. *E.g.*, **range**(1, 3, 0.5) generates the following list of reals: $[\,1.0, 1.5, 2.0, 2.5\,]$.

zeros(count) Generates a list consisting of `count` number of zeros. *E.g.*, **zeros**(4) generates $[\,0, 0, 0, 0\,]$.

Additionally, a number of mathematical functions are available to the programmer. These are listed below, and are essentially used in the same way as their equivalents are used in standard C/C++.

sin(x) Computes the sine of `x`, measured in radians.

cos(x) Computes the cosine of `x`, measured in radians.

tan(x) Computes the tangent of `x`, measured in radians.

asin(x) Computes the principal value of the arc sine of `x`.

acos(x) Computes the principal value of the arc cosine of `x`.

atan(x) Computes the principal value of the arc tangent of `x`.

atan2(x, y) Computes the principal value of the arc tangent of the quotient of `y` and `x`, using the signs of both arguments to determine the quadrant of the return value.

sinh(x) Computes the hyperbolic sine of `x`.

cosh(x) Computes the hyperbolic cosine of `x`.

tanh(x) Computes the hyperbolic tangent of x.

pow(x, y) Computes x raised to the power of y.

log(x) Computes the value of the natural logarithm of x.

log2(x) Computes the value of the logarithm of x to base 2.

log10(x) Computes the value of the logarithm of x to base 10.

sqrt(x) Computes the square root $\sqrt{x}$. If x is negative, nan (not a number) is returned.

exp(x) Computes the base-e exponential of x, *i.e.*, e^x.

As mentioned above, the eagerness of call-by-value languages, such as Fcc, implies that both the true-clause and the false-clause of applications of the `if`-function get evaluated. In some situations, this behaviour is undesirable, *e.g.*,

```
if(N == 10, 55, fib(N)).
```

In cases where such behaviour is undesirable, one can employ pattern matching on function signatures or make use of the optional condition-clause that is available in function definitions to ensure lazy evaluation.

A.4 Type System

The Fcc type system is static and implicit. In other words, variables and functions are implicitly typed during compile-time. The language has two base types, integers ($\mathbb{Z}$) and reals ($\mathbb{R}$). Additionally, the programmer can construct homogeneous lists from these two base types ($\mathbb{Z}^*$ and $\mathbb{R}^*$).

Well-defined function references can be assigned to variables and passed as arguments. Function types are deduced during compile-time, as described in §5.5 on page 44, and function applications are therefore fully determinable.

A.5 Function Definitions

Functions are defined as groups of one or more function clauses. Each function clause can be defined either with, (A.17), or without, (A.16), a conditional. Additionally, function signatures can embed pattern matching on integers and real numbers, and on the size of lists. The latter is described in §A.7.3 on page 88.

In Listing A.1, we find an example of a definition of a function group, `f`, containing three unique function clauses. Note the use of semicolons (on line one and two) to separate the clauses belonging to the same function group, and the use of period (on line three) to terminate the definition altogether. This function definition yields a function, `f`, which will return 0 if the first and only argument, N, is 1, N^2 if the argument is less than 10, and N otherwise.

Listing A.1: Pattern Matching and Conditionals in Function Definitions

```
1: f(1): 0;                # with pattern
2: f(N), N < 10: N * N;    # with conditional
3: f(N): N.                # catch-all clause
```

When the programmer implements functions with pattern-matching signatures and conditionals, the last clause of each function group must be a "catch-all" clause, meaning that the clause must match all possible combinations of input arguments that are not matched by any of the other clauses. The compiler checks whether the last clause satisfies this requirement. Additionally, the compiler complains if there exist multiple clauses catching the same input.

A.6 Code Blocks

In Fcc, code blocks are built up from one or more comma-separated statements. As can be deduced from the grammar specification in §A.1.2 on page 84, the language has only one occurrence of code blocks, namely in the representation of function bodies. It is worth noting that the last statement of the function body is considered to be the return statement of the function, *i.e.*, Fcc has no explicit return statements.

A.7 Lists and List Operations

Fcc supports homogeneous lists, either with elements of type integer ($\mathbb{Z}^*$) or real ($\mathbb{R}^*$). There exist two programming primitives that allow the programmer to construct and manipulate lists in addition to the pattern matching construct, (A.30), and the special-purpose functions mentioned in §A.3 on page 85, namely the list composition and list prefixation constructs.

A.7.1 List Composition

Lists are easily constructed, as in most other functional languages, by the use of square brackets, (A.53). Note that Fcc only deals with homogeneous lists. Hence, the programmer should ensure that all elements of a list are of the same type. If a list construction contains elements of both types, *e.g.*, as in Listing A.2, all elements will be converted into the dominant type , *i.e.*, into real numbers ($\mathbb{R}$).

Listing A.2: List Composition

```
List = [ 1, 2.2, 3, 4.1, 5 ] # ↦ [ 1.0, 2.2, 3.0, 4.1, 5.0 ]
```

A.7.2 List Prefixation

The programmer can prepend an element to an existent list using the prefixation operator, (A.54), as illustrated in Listing A.3.

Listing A.3: List Prefixation

```
List = [ 2, 3, 4 ], [ 1 | List ] # ↦ [ 1, 2, 3, 4 ]
```

Ordinary concatenation of two lists is easily implemented by the means of recursive prefixation of elements from the first list into the second list – see Listing A.4.

A.7.3 Pattern Matching

Fcc implements a primitive form of pattern matching for function signatures, (A.30), as shown in Listing A.4. Consider the first parameter of the `concat` function. We see that

there are two candidate matches, one for each function clause. The first candidate, `[]` on line two, matches all applications of `concat` where the first argument is the empty list. The second candidate, on line three, matches all non-empty lists and assigns the head of the list to `H` and the tail of the list to `T`. This is equivalent to naming the first parameter of `concat`, `L`, and assigning `head(L)` to `H` and `tail(L)` to `T`.

Listing A.4: List Concatenation

```
1: # concat : ℤ* × ℤ* ⟶ ℤ*
2: concat([],    L): L;
3: concat([H|T], L): [H|concat(T, L)].
4:
5: concat([ 1, 2, 3 ], [ 4, 5, 6 ]) # ↦ [ 1, 2, 3, 4, 5, 6 ]
```

APPENDIX B

Algorithms and Correctness

Contents

B.1 Optimisation of Augmenting Recursion

Definition A function, f, with k function clauses, $f_1, \ldots, f_k$, is said to be univariate if it has exactly one parameter. f is said to be multivariate if it has two or more parameters.

Definition In a univariate function, f, the parameter x_i denotes the instance of the parameter x in the function clause f_i. Syntactically, x_i is either a symbol or a pattern, the latter being a constant numerical value, an empty list or a list composition which is used in the matching of the function arguments of f (see §A.7.3 on page 88).

Definition A univariate function, f, is said to implement augmenting recursion and have one recursive case if:

- The last statement in exactly one of f's function clauses, name it f_k, is of the form $g\,(x_k, f(d(x_k)))$, where d is an invertible function[i], and g introduces no new base cases, is non-recursive and has no direct or indirect references to f.
- The parameter x_k is a symbol and all parameters x_i, $i \neq k$, are patterns.
- The last statement in all f_i, $i \neq k$, is of the form $a_i(x_i)$, where all a_i are trivial and satisfy the same criteria as g.

[i] The problem of deriving function inverses is, in itself, a separate research topic, as pointed out in [LS00].

Note that it makes no sense for a function to have more than one recursive case, as the pattern matching for these cases would overlap and result in an unpredictable program. However, functions can have multiple recursive calls, *e.g.*,

$$h = \lambda x.\ (h(x-1) + h(x-2))\,.$$

The employment of multiple recursive calls is not supported in FCC and is therefore ruled out from the definitions that are listed in this chapter.

Definition A multivariate function, f, with a set of parameters $x_{(1,i)}, x_{(2,i)}, \ldots, x_{(n,i)}$ for each function clause f_i, $1 \leq i \leq k$, is said to implement augmenting recursion and have one recursive case if:

- There is exactly one value for j which satisfies the following:
  - The parameter $x_{(j,k)}$ is a symbol.
  - All parameters $x_{(j,i)}$, $1 \leq i < k$, are patterns.
  - All parameters $x_{(p,i)}$, $p \neq j$ and $1 \leq i \leq k$, are symbols.
- The last statement in exactly one of f's function clauses, namely f_k, is of the form:

  $$g\left(x_{(1,k)}, \ldots, x_{(n,k)}, f(x_{(1,k)}, \ldots, x_{(j-1,k)}, d(x_{(j,k)}), x_{(j+1,k)}, \ldots, x_{(n,k)})\right),$$

  where d is an invertible function, and g introduces no new base cases, is non-recursive and has no direct or indirect references to f.
- The last statement in all f_i, $i \neq k$, is of the form:

  $$a_i(x_{(1,i)}, \ldots, x_{(n,i)}),$$

  where all a_i are trivial and satisfy the same criteria as g.

Definition A function, f, is said to have m base cases if it implements augmenting recursion, and if it has k function clauses and l recursive cases, where $k = m + l$.

Note that it is natural to assume that g is a polynomial function, or that g holds no operations which have the potential of breaking the flow and well-formedness of the program, such as, for instance, division which can raise an exception if the user tries to divide by zero. It should also be noted that recursive functions may not terminate, *e.g.*, consider a function with base case 0 and where d is defined to be $\lambda n.\ (n+2)$, or consider a function with base case 0, input argument -1, and where d is defined to be $\lambda n.\ (n-1)$.

Theorem B.1.1 *Univariate and multivariate functions which implement augmenting recursion and which have exactly one base case and one recursive case can be rewritten into tail-recursive form.*

Given a function, f, which implements augmenting recursion and which has exactly one base case and one recursive case, we need to show that Theorem B.1.1 holds. Thus, we need to consider both univariate and multivariate functions, and deal with arbitrary numbers of body statements for f. The complete proof can be extended to show the validity of similar transformations for functions with multiple base cases and for functions with multiple recursive calls in the recursive clause, as described in [LS00, BD77].

B.1.1 Univariate Functions

Lemma B.1.2 *All univariate functions which implement augmenting recursion and which have one base case and one recursive case, have equivalent tail-recursive functions.*

We will first consider single-statement functions, *i.e.*, functions with bodies which have no more than a single statement. The proofs extend trivially to multi-statement functions, as shown in §B.1.5 on page 95.

Proof Assume that we are given a univariate function, f, which implements augmenting recursion, and that this function has exactly one base case, f_1, and one recursive case, f_2. Also, assume that f_1 and f_2 have only one body statement each. Then, f_1 is of the form $a(\perp)$ and f_2 is of the form $g(x, f(d(x)))$.

Introduce two auxiliary parameters, t and r, and construct a multivariate function f', with parameters x, t and r. Let d' be the inverse function of d, meaning that $d'(d(x)) = d(d'(x)) = x$, and define f' as follows:

$$f'(x,t,r) = \begin{cases} g(x,r), & \text{if } x = t, \\ f'\left(d'(x), t, g(x,r)\right), & \text{otherwise.} \end{cases} \tag{B.1}$$

Rewrite f into a function, f'', of the form:

$$f''(x) = \begin{cases} a(\perp), & \text{if } x = \perp, \\ f'\left(d'(\perp), x, a(\perp)\right), & \text{otherwise.} \end{cases} \tag{B.2}$$

Since the transformation that is described in this section is not implemented in the current version of the prototype compiler, we confine ourselves to give only an outline of the proof of the equivalence relation between f and f''. The proof boils down to showing that the resulting call chains for the two functions are equivalent, *i.e.*, that:

$$\begin{aligned} f(x) &= g(x, g(d^{(1)}(x), g(d^{(2)}(x), g(\ldots, g(d^{(x-1)}(x), a(\perp))\ldots)))) && \text{(B.3)} \\ &= g(d'^{(x)}(\perp), g(d'^{(x-1)}(\perp), g(\ldots, g(d'^{(1)}(\perp), a(\perp))\ldots)))) && \text{(B.4)} \\ &= f''(x), && \text{(B.5)} \end{aligned}$$

where $d^{(k)}$ and $d'^{(k)}$ indicate k repeated applications of d and d', respectively.

Observe that f' is tail-recursive. Thus, by showing that f is equivalent to f'', through the use of the auxiliary function f', we show that there exists a tail-recursive function which is equivalent to the original function, f. This concludes the proof. □

B.1.2 Exploitation of Associativity and Commutativity

Definition A binary function, g, is associative iff the following axiom is satisfied:

$$g(a, g(b, c)) \stackrel{\text{def}}{=} g(g(a, b), c). \tag{B.6}$$

In our application, this extends trivially to n-ary functions, by the assumption that:

$$g(x_1, \ldots, x_k, a, g(x_1, \ldots, x_k, b, c)) = g(x_1, \ldots, x_k, g(x_1, \ldots, x_k, a, b), c), \tag{B.7}$$

where $k = n - 2$ and all x_i, $1 \leq i \leq k$, are constant. Associativity is then a matter of in which order the function g is applied to the arguments a, b and c. The arguments $x_1, \ldots, x_k$ are simply disregarded.

Definition A binary function, g, is commutative iff the following axiom is satisfied:

$$g(a,b) \stackrel{\text{def}}{=} g(b,a). \tag{B.8}$$

Similar to the above case of associative functions, the principle of commutativity can be extended to n-ary functions, by assuming constancy of parameters x_i, $1 \leq i \leq (n-2)$.

Lemma B.1.3 *If the operation, g, of a univariate function which implements augmenting recursion with one base case and one recursive case, is associative and commutative, and we have that $g(x, a(\perp)) = x$ for all x, then the properties of g can be exploited to simplify the tail-recursive function that we derived in §B.1.1.*

Proof Assume that we are given a univariate function, f, which implements augmenting recursion, and that this function has exactly one base case, f_1, and one recursive case, f_2. Also, assume that f_1 and f_2 have only one body statement each. Then, f_1 is of the form $a(\perp)$ and f_2 is of the form $g(x, f(d(x)))$, where g is both associative and commutative.

Introduce an auxiliary parameter, r, and define a new multivariate function f' as follows:

$$f'(x,r) = \begin{cases} r, & \text{if } x = \perp, \\ f'\left(d(x), g(r,x)\right), & \text{otherwise.} \end{cases} \tag{B.9}$$

Rewrite f into a function, f'', of the form:

$$f''(x) = f'(x, a(\perp)) \tag{B.10}$$

Now, we must prove the equivalence relation between f and f'', *i.e.*, we must show that $\forall x \geq \perp.\, f(x) = f''(x)$. However, the main computation is carried out by f'. Careful consideration about the process reveals that $f'(x,r)$ will apply g to r together with each of the values, $x, d^{(1)}(x), d^{(2)}(x), \ldots, d^{(x-1)}(x)$, where $d^{(k)}$ denotes k repeated applications of d. By the definition of f'', the initial value of r is given by $a(\perp)$. Hence, proving the equivalence relation between f and f'' corresponds to proving $f(x) = f''(x)$ by showing that $f'(x,r) = g(f(x), r)$, for $x \geq \perp$ and all r.

Lemma B.1.4 *For all $x \geq \perp$ and all r, $f'(x,r) = g(f(x), r)$.*

Proof By induction on x.

Base Case Observe that $f'(\perp, r) = r$, for all r, by the definition of f'. However, by the definition of f, $g(f(\perp), r) = g(a(\perp), r)$, which is equal to r, by commutativity and our initial assumption in Lemma B.1.3. So, $f'(\perp, r) = g(f(\perp), r)$. This concludes the base case.

Inductive Step Suppose that for some x and all r, $f'(x,r) = g(f(x), r)$. Now, we must show that $f'(d'(x), r) = g(f(d'(x)), r)$. Consider:

$$\begin{aligned}
f'(d'(x), r) &= f'(d(d'(x)), g(r, d'(x))), && \text{by the definition of } f'. \\
&= f'(x, g(r, d'(x))), && \text{by invertibility of } d. \\
&= g(f(x), g(r, d'(x))), && \text{by the induction hypothesis.} \\
&= g(f(x), g(d'(x), r)), && \text{by commutativity of } g. \\
&= g(g(f(x), d'(x)), r), && \text{by associativity of } g. \\
&= g(f(d'(x)), r), && \text{by the definition of } f.
\end{aligned}$$

As required, this shows that $f'(d'(x), r) = g(f(d'(x)), r)$ as long as we can assume that $f'(x, r) = g(f(x), r)$. So, this concludes the inductive step.

Proof of Lemma B.1.3, contd.
Let the predicate $P(n)$ be true if and only if $f(n) = f''(n)$. Then, we need to show that $P(\perp) \wedge (P(x) \rightarrow P(d'(x)))$ is true. However, Lemma B.1.4 proves that $f'(x, r) = g(f(x), r)$. So, to prove Lemma B.1.3, we only need to show that $f(x) = f'(x, a(\perp))$:

$$\begin{aligned} f''(x) &= f'(x, a(\perp)), && \text{by the definition of } f''. \\ &= g(f(x), a(\perp)), && \text{by Lemma B.1.4.} \\ &= f(x), && \text{by the initial assumption that } g(x, a(\perp)) = x. \end{aligned}$$

Observe that f' is tail-recursive. Thus, by showing that f is equivalent to f'', through the use of the auxiliary function f', we show that there exists a tail-recursive function which is equivalent to the original function, f. This concludes the proof. □

B.1.3 Multivariate Functions

Lemma B.1.5 *The transformation techniques that were presented in §B.1.1 and §B.1.2 extend trivially to multivariate functions and, hence, can turn instances of augmenting recursion in multivariate functions into tail-recursion.*

Proof Observe that, by fixing x_k, $1 \leq k < n$, in all applications of $f_{(n)}$, we can easily construct f, such that:

$$f_{(1)}(x_n) = f_{(n)}(x_1, \ldots, x_{(n-1)}, x_n), \tag{B.11}$$

for all x_n. This correlates to introducing constants into $f_{(1)}$. The previously presented transformation techniques require that g and a are extended in a similar manner, to take the additional arguments, $x_1, \ldots, x_{(n-1)}$. Note that the extension of g does not affect the associativity and commutativity of the function as long as all x_i, $1 \leq i \leq (n-1)$, are constant – see definitions of associative and commutative functions in §B.1.2. □

We have shown that both univariate and multivariate functions which implement augmenting recursion, with exactly one base case and one recursive case, can be converted into equivalent tail-recursive functions. This concludes the proof of Theorem B.1.1. □

B.1.4 Structural Recursion

Lemma B.1.6 *Structural recursion over one-dimensional data structures,* e.g.*, the traversal over homogeneous lists, is an instance of augmenting recursion, and can be converted into an equivalent tail-recursive form, given that the identified function, g, is associative.*

Proof Structural recursion can be regarded as an element-wise traversal over data structures. Consequently, in cases where we are dealing with one-dimensional data, structural recursion is an instance of augmenting recursion where we iteratively perform some action on the first element of the structure, the head, together with the result that is acquired from processing the remainder of the structure, the tail:

$$f(x) = \begin{cases} a([\,]), & \text{if } x = [\,], \\ g(x, f([\,l_2, \ldots, l_n\,])), & \text{if } x = [\,l_1, l_2, \ldots, l_n\,]. \end{cases} \tag{B.12}$$

The empty list is clearly identified as the base case, $\perp = [\,]$, and d is defined to return the tail of the provided list, *i.e.*, $d([\,l_1, l_2, \ldots, l_n\,]) = [\,l_2, \ldots, l_n\,]$. This allows us to rewrite the second case of the function, f, in (B.12) into $g(x, f(d(x)))$.

Now, let $g(x, r)$ be a function of the form $\lambda x.\,(h(x) \bullet r)$, where $\bullet$ is associative and h returns the first element of the supplied list, x. Note that the definition of g makes the function associative, but not necessarily commutative.

Since g is not guaranteed to be commutative, we need to take a slightly different approach to the one that was presented in §B.1.2. More specifically, we need to introduce a few minor changes to f' and f''. As before, introduce an auxiliary parameter, r, and define a new multivariate function f' as follows:

$$f'(x, r) = \begin{cases} g(r, a(\perp)), & \text{if } x = \perp, \\ f'\left(d(x), g(r, x)\right), & \text{otherwise.} \end{cases} \tag{B.13}$$

Also, rewrite the original function, f, into a function, f'', of the form:

$$f''(x) = \begin{cases} a(\perp), & \text{if } x = \perp, \\ f'\left(d(x), x\right), & \text{otherwise.} \end{cases} \tag{B.14}$$

We will not give a complete proof of this transformation, but, by using the associativity of g, it can be shown that f and f'' are equivalent. As before, this essentially boils down to showing that the resulting call chains for the two functions are equivalent:

$$\begin{aligned} f(x) &= g(x, g(d^{(1)}(x), g(d^{(2)}(x), g(\ldots, g(d^{(x-1)}(x), a(\perp))\ldots)))) && \text{(B.15)} \\ &= g(g(g(\ldots g(g(x, d^{(1)}(x)), d^{(2)}(x)), \ldots), d^{(x-1)}(x)), a(\perp)) && \text{(B.16)} \\ &= f''(x), && \text{(B.17)} \end{aligned}$$

where $d^{(k)}$ indicates k repeated applications of d. Notice the equivalence between (B.15) and (B.16), which is a direct result of the fact that $g(a, g(b, c)) = g(g(a, b), c)$. □

B.1.5 Mutual Recursion and Multi-Statement Bodies

Definition A function, h is said to have multiple body statements, or a multi-statement body, if and only if h is of the form:

$$h(x) = \{\, t_1 \leftarrow s_1;\, t_2 \leftarrow s_2;\, \ldots;\, t_k \leftarrow s_k;\, \}, \tag{B.18}$$

where t_i can appear in one or more s_j, $i < j \leq k$.

Definition Two mutually recursive functions, e and f, are called well-behaved if and only if they are of the form:

$$\begin{aligned} e(x) &= f(d(x)) && \text{(B.19)} \\ f(x) &= \begin{cases} a(\perp), & \text{if } x = \perp, \\ g(x, e(x)), & \text{otherwise.} \end{cases} && \text{(B.20)} \end{aligned}$$

Lemma B.1.7 *The transformation of augmenting recursion into tail recursion extends naturally to well-behaved mutually recursive functions and functions with multiple body statements.*

Proof Observe that functions, h, with multiple body statements can be written as:

$$h(x) = h'(s_1, (\lambda t_1.\ s_2), (\lambda t_1 \lambda t_2.\ s_3), \ldots, (\lambda t_1 \lambda t_2 \ldots \lambda t_{(k-1)}.\ s_k)), \tag{B.21}$$

with the function, h', defined as follows:

$$h'(u_1, u_2, \ldots, u_k) = u_k(u_1, u_2(u_1), u_3(u_1, u_2(u_1)), \ldots, u_{(k-1)}(\ldots)). \tag{B.22}$$

This abstract transformation yields well-behaved mutually recursive functions. Thus, for the concepts that have been presented previously to apply to functions with multiple body statements, the concepts must also extend to well-behaved mutually recursive functions. It is routine to show that this is indeed the case, and to verify that the concepts extend to well-behaved mutual recursion. □

APPENDIX C

Language Models

Contents

C.1 Type Model

This section defines the type model that is used in the type inference phase of the compilation process in Fcc. The type inference system tries to derive a solution satisfying these criteria. If the type rules cannot be satisfied, the compilation will fail since there, by definition, exist no valid type assignments for the input program.

The language is built up from the following base types:

$$B \quad ::= \quad \mathbb{Z} \quad | \quad \mathbb{R} \quad | \quad \mathbb{Z}^* \quad | \quad \mathbb{R}^* \tag{C.1}$$

The set of types that are available to the programmer is recursively defined as follows:

$$T \quad ::= \quad 1 \quad | \quad B \quad | \quad T \longrightarrow T' \tag{C.2}$$

The following bullet points define the typing rules for the Fcc language:

- Function Group
  Each function group is a collection of n function clauses, λ_i, for $1 \leq i \leq n$, such that the following type assignment is satisfied:

$$\frac{\lambda_1 : T \quad \ldots \quad \lambda_n : T}{\lambda = \{\lambda_1, \ldots, \lambda_n\} : T} \tag{C.3}$$

  In other words, all the function clauses of the function group are expected to be of the same type, T.

- Function
  For a function with m parameters, $x_1, \ldots, x_m$, and n body statements, $t_1, \ldots, t_n$, the following type rule must be satisfied:

$$\frac{x_1 : T_1 \quad \ldots \quad x_m : T_m \qquad\qquad t_1 : T'_1 \quad \ldots \quad t_n : T'_n}{\lambda x_1 \ldots \lambda x_m.\ (t_1, \ldots, t_n) : T_1 \times \cdots \times T_m \longrightarrow T'_n}, \tag{C.4}$$

  where $T_1 \times \cdots \times T_m$ is an alternative notation for:

$$T_1 \longrightarrow T_2 \longrightarrow \ \cdots \ \longrightarrow T_m.$$

  The latter is the normal form in λ-calculus, where each function has a single parameter, and a set of λ-functions is needed to represent a function with multiple parameters.

- Scalars
  All numbers are either of type integer, $\mathbb{Z}$, or real, $\mathbb{R}$. The initial type of the number is determined from whether it is written in a decimal or an exponential form, or as an integer.

- Vectors
  Vectors are homogeneous, meaning that all the elements must be of the same type. Therefore, the following type rule must be satisfied:

$$\frac{t_1 : T \quad \ldots \quad t_n : T}{[\, t_1, \ldots, t_n \,] : T^*} \qquad \text{where } T \text{ is either } \mathbb{Z} \text{ or } \mathbb{R}. \tag{C.5}$$

- Operations
  The following rules describe the type assignment of operands in arithmetic and logical operations.

  - Addition, Subtraction and Multiplication
    These arithmetic operations simply use the most dominant type, T, of t_i, $1 \leq i \leq n$. During compilation, some t_i might be of different types, but after the type deduction stage, the least dominantly typed terms will obtain the type of the term whose type has the highest rank.

$$\frac{t_1 : T \quad \ldots \quad t_n : T}{t_1 \bullet \cdots \bullet t_n : T} \qquad \text{where } \bullet \text{ is } +, -, \text{ or } \times, \text{ and } T \text{ is } \mathbb{Z} \text{ or } \mathbb{R}. \tag{C.6}$$

  - Unary Operations
    Unary operations, such as arithmetic and logical negation, do not affect the type assignment of the operand.
  - Division
    There are two kinds of division available in FCC, namely integer division and real division. The latter is always of type real, regardless of the types of t_1 and t_2. Observe that if a symbol of type integer is used in a real division, the type of the symbol will be converted into real.

$$\frac{t_1 : \mathbb{R} \qquad t_2 : \mathbb{R}}{t_1 \div t_2 : \mathbb{R}} \qquad\qquad \frac{t_1 : \mathbb{Z} \qquad t_2 : \mathbb{Z}}{\lfloor t_1 \div t_2 \rfloor : \mathbb{Z}} \tag{C.7}$$

  - Modulus
    The modulus operator computes the remainder of an integer division.

$$\frac{t_1 : \mathbb{Z} \qquad t_2 : \mathbb{Z}}{t_1 \text{ rem } t_2 : \mathbb{Z}} \tag{C.8}$$

  - Boolean Arithmetic
    All boolean operations, such as numerical comparisons, and logical conjunctions, disjunctions and negations, return values of type integer. The logical operators expect operands of integral types. However, the operand types are not affected for numerical comparisons.

- Function Application
  Assuming that the function f is defined and that we have terms $t_1, \ldots, t_k$ of corresponding types, the following type rule holds:

$$\frac{f : T_1 \times \cdots \times T_k \longrightarrow T \qquad t_1 : T_1 \quad \ldots \quad t_k : T_k}{f(t_1, \ldots, t_k) : T} \tag{C.9}$$

- List Prefixation and List Composition
  The prefixation of elements to a list yields a list of the same type as the original. Note, however, that we expect consistency between the types of t_1 and t_2. List composition expects consistency between the types of all t_i, for $1 \leq i \leq n$.

$$\frac{t_1 : T \qquad t_2 : T^*}{[\, t_1 \,|\, t_2 \,] : T^*} \qquad\qquad \frac{t_1 : T \quad \ldots \quad t_n : T}{[\, t_1, \ldots, t_n \,] : T^*} \tag{C.10}$$

- Pattern Matching
  When we perform pattern matching on lists, we expect the matched head and tail entities to have consistent types. That is to say that the head and the tail always obtain types corresponding to the input list.

$$\frac{[\, t_1 \,|\, t_2 \,] : T^*}{t_1 : T \qquad t_2 : T^*} \tag{C.11}$$

Type	prec(T)	Description
$\mathbb{Z}$	1	Integers
$\mathbb{R}$	2	Reals
$\mathbb{Z}^*$	3	Lists of integers
$\mathbb{R}^*$	4	Lists of reals
$\mathcal{F}$	5	Function literals

Table C.1: Type precedence, sorted from lowest to highest.

The function in (C.12) describes the `Type.dominion`-function that is implemented in the `Type`-class in "`semantics/types.py`."

$$\texttt{dom}(T_1, T_2) = \begin{cases} T_1 & \text{if prec}(T_1) \geq \text{prec}(T_2), \\ T_2 & \text{otherwise.} \end{cases} \tag{C.12}$$

The `dom`-function extends trivially to more than two parameters. $\texttt{dom}^*$ recursively finds the most dominant type from a list of types. Note that we assume that n is strictly ≥ 2.

$$\texttt{dom}^*(T_1, \ldots, T_n) = \begin{cases} \texttt{dom}(T_1, T_2) & \text{if } n = 2, \\ \texttt{dom}(T_1, \texttt{dom}^*(T_2, \ldots, T_n)) & \text{otherwise.} \end{cases} \tag{C.13}$$

C.2 Performance Model

This section defines the performance model that is used by Fcc to determine whether a loop should be parallelised or not. The implementation of the model can be found in "`analysis/complexity.py`." Fcc applies a heuristic function, based on this performance model, to all the loops of the input program to estimate the workload of the corresponding loop bodies. The heuristic function traverses all the branches of the syntax tree that are associated with the considered loop bodies, and returns a measure of the computational cost of each loop. The resulting measures are used at run-time, together with the size of each loop's input data, to determine whether a candidate should be run in parallel or not.

Note that the model that is described in this section assumes that the intermediate code representation is in a format similar to three-address code.

Arithmetic Operations

The cost of arithmetic operations is based on the product of a constant and the number of operands:

$$\mathcal{P}[\![\, \mathtt{x}_1 \bullet \cdots \bullet \mathtt{x}_n \,]\!] = n - 1 \qquad \text{where } \bullet \text{ is + or -.} \tag{C.14}$$

$$\mathcal{P}[\![\, \mathtt{x}_1 \bullet \cdots \bullet \mathtt{x}_n \,]\!] = 2(n - 1) \qquad \text{where } \bullet \text{ is } *, \backslash \text{ or } \%. \tag{C.15}$$

$$\mathcal{P}[\![\, \mathtt{x}_1 \ / \ \mathtt{x}_2 \,]\!] = 3 \tag{C.16}$$

Boolean operations and comparisons have cost $n - 1$, for n operands. List prefixations are more costly and have cost $4(n - 1)$. However, since the number of operands for the prefixation operation is bound to two, the cost is constant.

Note that the operands in the operations that are described above are bound to be either numbers or variables. The costs that are normally associated with memory accesses are different from the ones that are associated with the use of immediate values. However, Fcc does not discriminate between the two. Since Fcc deals with high-level constructs and since the back-end produces C code, the compiler cannot know for sure whether a memory access is indeed a memory access or if the underlying C compiler converts the memory access into a register access.

List Composition

The cost of list composition is defined to be two times the size of the list. So for a composition of n elements, the associated cost would be $2n$.

Assignments

The computational cost of assignments are static, and only depends on the type of the destination. Integer and real assignments are cheaper than list assignments, as shown below:

$$\mathcal{P}[\![\, \mathtt{x}_d \ \mathtt{=}\ \mathtt{term} \,]\!] = 2 + \mathcal{P}[\![\, \mathtt{term} \,]\!] \qquad \text{where } \mathtt{x}_d \text{ is of type } \mathbb{Z}^* \text{ or } \mathbb{R}^*. \tag{C.17}$$

$$\mathcal{P}[\![\, \mathtt{x}_d \ \mathtt{=}\ \mathtt{term} \,]\!] = 1 + \mathcal{P}[\![\, \mathtt{term} \,]\!] \qquad \text{where } \mathtt{x}_d \text{ is of type } \mathbb{Z}, \mathbb{R} \text{ or } \mathcal{F}. \tag{C.18}$$

Function Applications

All the special-purpose functions that were listed in §A.3 on page 85 have a pre-defined cost associated with them. Otherwise, if the call-target is unknown, a default penalty of 10 is imposed on the function call.

Function	Cost
`if`	3
`head`	1
`tail`	8
`len`	1
`empty`	1
`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `pow`, `log`, `log10`, `log2`, `exp`, `sqrt`	5
`range(`f`, `t`, `s`)`[i]	$1.5(t-f)/s$
`zeros(`n`)`[i]	n

Table C.2: Computational cost of special-purpose functions.

In further work, effort should be put into making the compiler perform more mature estimates for applications of unknown functions, *e.g.*, by trying to deduce the call-target at compile-time and estimating the cost of application accordingly.

For- and While-Loops

Loops are not available to the programmer, but they are used by the compiler in the intermediate code representation to store the structure of instances of recursion that have been converted into iteration. The cost of `for`-loops is modelled by:

$$\mathcal{P}[\![\texttt{for (i = } f\texttt{; i != } t\texttt{; i += } s\texttt{) \{ body \}}]\!] = (\delta + \mathcal{P}[\![\texttt{body}]\!]) \times \frac{t-f}{s}, \quad \text{(C.19)}$$

where δ is a pre-defined iteration overhead. Similarly, the cost of `while`-loops is modelled by:

$$\mathcal{P}[\![\texttt{while (cond) \{ body \}}]\!] = (\mathcal{P}[\![\texttt{cond}]\!] + \mathcal{P}[\![\texttt{body}]\!]) \times 25. \quad \text{(C.20)}$$

Unknown Iteration Counts

When statically estimating the computational cost of looping constructs, *i.e.*, in cases where the cost of the loop has to be determined at compile-time[ii], we are likely to run into situations where the iteration count is unknown. In these cases, we assume an iteration count of 25 [KY04]. The iteration counts of internally represented `while`-loops, *i.e.*, (C.20), are always unknown.

[i] If the input arguments are unknown, meaning non-constant, a default penalty of 64 is imposed on the function call.

[ii] For instance, since CUDA disallows nested parallelism, loops inside the body of a CUDA kernel cannot be parallelised and must be regarded as sequential units which get run for every instance of the CUDA kernel.

APPENDIX D

Implementation

Contents

D.1 Usage Information

The compiler can be run by issuing the `fcc` command from the compiler's root directory, assuming that the Python interpreter is set up properly, and that the path of the `python3` executable is listed in the `PATH` environment variable. Similarly, to compile the generated output with external build tools, the `nvcc` and `gcc` executables must reside in one of the directories that are listed in the `PATH` variable.

A set of compilation options are available to the user, and these can be listed by issuing the `fcc` command without supplying any command line arguments. The available options are described in detail in "`options.py`."

D.2 Source Code Hierarchy

This appendix presents the source code hierarchy of the implemented prototype compiler. A brief description of what features can be found in each of the source files is provided in the table below. The implemented solution, together with an electronic copy of this document, can be found on the enclosed CD.

`analysis/complexity.py`	Implementation of the performance cost model that is used in the parallelisation stage of the compilation process.
`analysis/dependence.py`	Implementation of the dependence analyses and the various analytic tools that are used in the identification of induction variables and recursive structures.
`backend/__init__.py`	Implementation of the code generation system, and the abstract classes that are essential for emitting output code and invoking external build tools.
`backend/c/__init__.py`	Implementation of the code emitter for sequential C99 code.
`backend/c/lib.py`	Implementation of the system which dynamically generates library functions for C99 output code.
`backend/c/make.py`	Implementation of the component which is responsible for invoking the external GCC build system, and for processing the results.
`backend/c/static.py`	Listing of the strings that are used in the generation of C99 library code.
`backend/cuda/__init__.py`	Implementation of the code emitter for parallel CUDA C/C++ code.
`backend/cuda/lib.py`	Implementation of the system which dynamically generates library functions for CUDA C/C++ output code.
`backend/cuda/make.py`	Implementation of the component which is responsible for invoking the external CUDA compiler, and for processing the results.
`backend/cuda/static.py`	Listing of the strings that are used in the generation of CUDA library code.
`optimisation/general.py`	Implementation of the commonly applied compiler transformations.

`optimisation/interproc.py`	Implementation of the functionality that is used in the interprocedural optimisations, *e.g.*, function inlining.
`optimisation/recursion.py`	Implementation of the functionality that is used in transforming recursion into iteration.
`compiler.py`	Implementation of the component that links all the compilation stages together, and which deals with error handling, *etc.*
`fcc`	The program entry-point and implementation of functionality for processing command-line arguments.
`options.py`	Encapsulation of all the user options that are related to the compilation and parallelisation processes.
`parser/lex.py`	Implementation of the lexical analyser.
`parser/yacc.py`	Implementation of the syntactic analyser.
`semantics/feedback.py`	Implementation of the functionality that is used to keep track of source code locations and to print error and warning messages.
`semantics/ssa.py`	Implementation of the functionality that is used to convert the abstract syntax tree into three-address code[i], and back again.
`semantics/tree.py`	Representation of the abstract syntax tree, and the symbol table that is used by the front-end and the middle-end.
`semantics/types.py`	Representation of the types that are supported by the Fcc language.
`tools/dot.py`	Tool for translating the abstract syntax tree into a Graphviz[ii] graph.
`unittests.py`	Unit tests for the implemented solution.

[i] Static single assignment form (SSA) is a refinement of three-address code.
[ii] `http://www.graphviz.org/`

D.3 Code Excerpts

The following subsections contain code listings for a selection of the classes and functions that can be found in the implementation of the Fcc prototype compiler. The code excerpts are sorted by the names of the source files in which they exist. Omitted code sections are denoted by gray, shaded rectangles.

D.3.1 optimisation/general.py

```
import optimisation.interproc as interproc
from options import Options
from semantics.ssa import is_temporary, temporary
from semantics.tree import *

class Optimiser:
    '''Implements a set of optimisation techniques applicable to
    abstract syntax trees in static single-assignment form.'''

    input_has_changed = False

    def __init__(self, techniques = None):
        '''Instantiates a new Optimiser object. The 'techniques'
        parameter can be used to specify what techniques to apply to
        the input. If 'techniques' is equal to None, the set of
        techniques is determined from the options provided by the
        Options class.'''
        self.has_changed = False
        self.techniques = []
        self.searched = []
        if not Options.use_ssa:
            return # only accept input in ssa form
        self.techniques = techniques

    def traverse(self, node, callback, reset_callback=None):
        '''Traverses all statements of the abstract syntax tree and
        invokes the callback function on each occurrence.'''
        if node == None:
            return
        elif node.match(Combinator):
            for statement in node.data.list:
                if statement != None:
                    self.traverse(statement, callback, reset_callback)
                    callback(statement)
        elif node.match(Function):
            # set local environment
            Symbol.cache = node.data.symbol.data.symbol().environment
```

```
            self.traverse(node.data.condition, callback, reset_callback)
            if node.data.condition != None:
                callback(node.data.condition)
            self.traverse(node.data.body, callback, reset_callback)
            callback(node.data.body)
            # reset local environment
            Symbol.cache = {}
            callback(node)
        elif node.match(Operation):
            for operand in node.data.operands:
                self.traverse(operand, callback, reset_callback)
                callback(operand)
        elif node.match(List):
            for element in node.data.list:
                self.traverse(element, callback, reset_callback)
                callback(element)
        elif node.match(Assignment):
            self.traverse(node.data.rhs, callback, reset_callback)
            callback(node.data.rhs)
        elif node.match(FunctionApplication):
            self.traverse(node.data.function, callback, reset_callback)
            callback(node.data.function)
            for argument in node.data.arguments:
                self.traverse(argument, callback, reset_callback)
                callback(argument)
```

```
    def common_subexpression_elimination(self):
        '''Performs common subexpression elimination during the
        optimisation stage, e.g.,
        t1 = X - 1, t2 = X - 1 -> t1 = X - 1, t2 = t1.'''
        collected_terms = {}
        def process(node):
            if node.match(Assignment) and not node.is_untouchable:
                # ignore untouchable symbols
                if node.data.lhs.match(SymbolReference):
                    sym = node.data.lhs.data.symbol()
                    if sym != None and sym.is_untouchable:
                        return
                if node.data.rhs in collected_terms.values():
                    # reuse of subexpression ->
                    #     reference old assignment
                    sym = find_key(collected_terms, node.data.rhs)
                    sym = Node(SymbolReference(sym))
                    node.substitute(Node(Assignment(node.data.lhs, sym)))
                    self.has_changed = True
```

```
                else:
                    # does not exist -> keep a record of subexpressions
                    collected_terms[node.data.lhs.data.name] = \
                            node.data.rhs
        def reset(node):
            nonlocal collected_terms
            collected_terms = {} # forget state before proceeding; after
                                 # having processed a loop body
        self.traverse(self.root, process, reset)
```

D.3.2 optimisation/interproc.py

```
from semantics.tree import *
from semantics.ssa import temporary
from analysis.dependence import symbol_references
```

```
def inline_functions(functions):
    '''Performs procedure inlining for small leaf functions. This can
    help the compiler to identify the nature of operations on induction
    variables in recursive functions. E.g., consider the function
    dec(N): N - 1, and the recursive function: f(N): N * f(dec(N)).'''
    global active_function_group
    active_function_group = functions
    candidates, has_changed = {}, False

    # collect candidates for function inlining, i.e., all trivial
    # leaf functions
    all_patterns = []
    for k in functions:
        if is_trivial_leaf_function(functions[k]):
            symbol = functions[k][0].data.signature()
            if not symbol.is_untouchable:
                candidates[symbol] = functions[k][0]
        patterns = trivial_signatures(functions[k])
        for pattern in patterns:
            all_patterns.append(pattern)

    # substitute all applications of candidate functions
    for k in functions:
      for function in functions[k]:
```

```
Symbol.cache = function.data.signature().environment
# collect all relevant applications in given function body
function_applications = collect(
     lambda n:
         n.__class__ == FunctionApplication and
         not n.function.data.symbol().is_untouchable and (
         n.function.data.symbol() in candidates or
         n.function.data.symbol() in
             [c.data.signature() for c in all_patterns]),
     function.data.body)
for application in function_applications:
  # inline function
  if application.function.data.symbol() in candidates:
      candidate = candidates[application.function.data.symbol()]
  else:
      # match against all available patterns
      candidate = None
      for c in [c for c in all_patterns
                  if c.data.signature() ==
                      application.function.data.symbol()]:
          matching = True
          for i in range(len(application.arguments)):
              arg = application.arguments[i]
              param = c.data.parameters[i]
              if arg.match(SymbolReference) or \
                      param.match(SymbolReference):
                  matching = False
                  break
              if arg != param:
                  matching = False
                  break
          if matching:
              candidate = c
              break
      if candidate == None:
          continue
  substitution = copy(candidate)
  syms = [] # keep track of argument symbols that have been
            # substituted in to the inlined code block
  for i in range(len(application.arguments)):
      if application.arguments[i].match(SymbolReference):
          syms.append(application.arguments[i].data.symbol())
  new_syms = []
  # replace locally used symbols to avoid duplication if
  # multiple instances of the same function are inlined in
  # the same code block
  collected_syms = {}
  for sym in set(collect(lambda n:
          n.__class__ == Node and n.match(SymbolReference),
          substitution)):
```

```
            if sym.data.name[0].isupper() and \
                    sym not in collected_syms:
                new_sym = temporary()
                new_sym.typify(sym.type())
                collected_syms[sym] = new_sym
        for sym in collected_syms:
            new_sym = collected_syms[sym]
            substitution = replace(substitution, sym, new_sym)
            new_syms.append(new_sym.data.symbol())
        # substitute in the passed arguments for the parameters
        for i in range(len(application.arguments)):
            if candidate.data.parameters[i].match(SymbolReference):
                psym = candidate.data.parameters[i]
                if psym in collected_syms:
                    psym = collected_syms[psym]
                asgn = Node(Assignment(psym,
                        application.arguments[i]))
                substitution.data.body.data.list.insert(0, asgn)
        substitution = substitution.data.body
        # perform the substitution
        if len(substitution) == 1:
            application.substitute(substitution[0])
        else:
            application.substitute(substitution)
        has_changed = True
        # collect new symbols and add to environment
        for sym in set(collect(
                lambda n: n.__class__ == SymbolReference,
                substitution)):
            if sym.name not in function.data.signature().environment \
                    and sym.name.find('') == -1:
                function.data.signature().environment[sym.name] = \
                        sym.symbol()
        # add all new temporaries to the environment
        for sym in set(new_syms):
            function.data.signature().environment[sym.name] = sym
        break   # only do one at the time (allow for parent node to
                # update before proceeding)
    return (functions, has_changed)
```

D.3.3 optimisation/recursion.py

```
from semantics.tree import *
from semantics.ssa import temporary
from analysis.dependence import *
```

```
from optimisation.general import Optimiser

def find_induction_variable(function):
    '''Deduces the induction variable in a function group. Returns
    'None' if none is found.'''
    p = function[-1].data.parameters
    results = []
    for i in range(len(p)):
        if p[i].match(SymbolReference) or p[i].match(Operation):
            for j in range(len(function)-1):
                if function[j].data.parameters[i].match(Number):
                    if (p[i], i) not in results:
                        results.append((p[i], i, None))
                elif function[j].data.parameters[i].match(List):
                    if (p[i][1], i) not in results:
                        results.append((p[i][1], i, p[i][0]))
    if len(results) == 1:
        return results[0] # we can only deal with cases where we
                          # have a single induction variable
    else:
        return (None, -1, None)
```

D.3.4 semantics/tree.py

```
class Node:
    '''Object for holding instances of subclasses of DataNode. This
    simplifies substitution and manipulation of nodes in the abstract
    syntax tree during the generation and optimisation stages of the
    compilation.'''

    has_changed = False
    '''Used to indicate changes to the inferred type system.'''

    @staticmethod
    def amend(old_type, new_type):
        '''Flags the type system as changed if the two specified
        types differ. The 'Node.has_changed' flag is used to determine
        whether we have reached a stable typing state for the input
        program or not.'''
        if old_type != new_type: Node.has_changed = True

    @staticmethod
    def compare(a, b, selector):
        '''Compares two objects based on the result of the selector
        function.'''
```

```
        if a.__class__ == b.__class__:
            o1, o2 = selector(a), selector(b)
            if o1.__class__ == o2.__class__:
                if o1 < o2:     return -1
                elif o1 == o2:  return  0
                else:           return +1
        # objects of different classes cannot be equal
        return -1

    def __init__(self, data):
        '''Initialises a new Node object. Nodes work as slots for
        subclasses of the DataNode class. This allows us to easily
        substitute nodes in the optimisation stage of the compilation
        process without keeping track of parent nodes.'''
        self.data = copy(data)
        self.data.parent = self
        self.is_untouchable = False
        if hasattr(self.data, '__iter__'):
            for element in self.data:
                element.parent = self.data
        self.parent = None
        self.update()

    def __lt__(self, other):
        '''self < other <=> self.__lt__(other)'''
        return Node.compare(self, other, lambda o: o.data) < 0

    def __eq__(self, other):
        '''self == other <=> self.__eq__(other)'''
        return Node.compare(self, other, lambda o: o.data) == 0

    def __repr__(self):
        '''Returns a string representation of the Node object.'''
        return repr(self.data)

    def __iter__(self):
        '''Returns an iterator for the Node object. Used to iterate
        over statement lists, list objects, etc.'''
        if self.data != None and len(self.data) > 0:
            for element in self.data: yield element

    def __hash__(self):
        '''Obtains a hash key for the current object.'''
        return hash(self.__repr__())
```

```
    def type(self):
        '''Deduces the type of the Node object.'''
        return self.data.type()

    def typify(self, type):
        '''Puts new constraints on the type of the Node object.'''
        self.data.typify(type)

    def substitute(self, new):
        '''Substitutes the data associated with this node with the data
        associated with another node. The substitution is a matter of
        redirecting the 'data' slot to a new object.'''
        new_data = copy(new.data)
        self.data = new_data
        self.data.parent = self
        self.update()
        return self

    def __getitem__(self, index):
        '''x.__getitem__(y) <==> x[y]'''
        return self.data.__getitem__(index)

    def __setitem__(self, index, value):
        '''x.__setitem__(i, y) <==> x[i]=y'''
        return self.data.__setitem__(index, value)

    def __len__(self):
        '''x.__len__() <==> len(x)'''
        return len(self.data)

    def update(self):
        '''Calls the 'update' slot of the DataNode. Used to simplify/
        optimise branches based on their structure.'''
        if hasattr(self.data, 'update'):
            self.data.update()
        if self.parent and hasattr(self.parent, 'update'):
            self.parent.update()

    def match(self, cls):
        '''Shorthand method to check the class of the associated
        DataNode object.'''
        return self.data.__class__ == cls

    def error(self, format, *args):
        '''Prints an error message based on the source code location
        associated with the node.'''
        self.data.error(format, *args)

    def warning(self, level, format, *args):
        '''Prints a warning message based on the source code location
```

```
        associated with the node.'''
        self.data.warning(level, format, *args)
```

```
class NodeList:
    '''An extension of the 'list' class to allow for tracking of the
    parent list for all elements of the list. This is mainly used in
    the process of simplifying instances of the Operation and Combinator
    classes where changes to the list itself might be desirable after a
    simplification or modification to one of its elements..'''

    def __init__(self, parent, *elements):
        '''Instantiates a new NodeList object from 'elements'. 'parent'
        refers to the owner node.'''
        self.parent = parent
        if len(elements) == 1 and type(elements) is tuple:
            elements = elements[0]
        self.elements = []
        for element in elements:
            new_element = copy(element)
            new_element.parent = self.parent
            self.elements.append(new_element)

    def __lt__(self, other):
        '''self < other <=> self.__lt__(other)'''
        return Node.compare(self, other, lambda o: o.elements) < 0

    def __eq__(self, other):
        '''self == other <=> self.__eq__(other)'''
        return Node.compare(self, other, lambda o: o.elements) == 0

    def __getitem__(self, index):
        '''x.__getitem__(y) <==> x[y]'''
        return self.elements[index]

    def __getslice__(self, i, j):
        '''x.__getslice__(i, j) <==> x[i:j]'''
        return self.elements[i:j]

    def __len__(self):
        '''x.__len__() <==> len(x)'''
        return len(self.elements)

    def __repr__(self):
        '''Returns a string representation of the list.'''
        return '␣'.join([str(e) for e in self.elements])
```

```
    def __iter__(self):
        '''Returns an iterator for the list.'''
        for element in self.elements:
            yield element

    def sort(self):
        '''Performs an inline sort on the elements in the list.'''
        self.elements.sort()

    def append(self, element):
        '''Appends 'element' to the end of the list.'''
        new_element = copy(element)
        new_element.parent = self.parent
        self.elements.append(new_element)

    def insert(self, index, element):
        '''Inserts 'element' into the list before the element at
        'index'.'''
        new_element = copy(element)
        new_element.parent = self.parent
        self.elements.insert(index, new_element)

    def remove(self, element):
        '''Removes 'element' from the list.'''
        self.elements.remove(element)
```

```
class DataNode:
    '''Allows for the creation of data objects which are used to store
    all the unique node information (i.e., non-shared attributes) in the
    abstract syntax tree. The 'data' slot in the Node class can hold any
    object which derives from of DataNode.'''

    def __init__(self):
        '''Instantiates a new DataNode - all derived subclasses should
        invoke this constructor.'''
        self.parent = None
        self.location = Location()
        self.cached_type = Type.Int

    def update_type(self, type):
        '''Used to check whether a newly derived type implies changes
        to the deduced type system. If so, the type system gets flagged
        as changed and the cached type for this particular DataNode
        instance gets updated.'''
```

```
        Node.amend(self.cached_type, type)
        self.cached_type = type
        return self.cached_type

    def type(self):
        '''Deduces the type of this particular DataNode instance. This
        method should be overridden by all classes deriving from
        DataNode.'''
        return Type.Void

    def __lt__(self, other):
        '''self < other <=> self.__lt__(other)'''
        return -1

    def __getitem__(self, index):
        '''Used as a means to exploit indexed elements of a node if the
        node is of an n-ary nature or if it employs a list structure.
        This method should be overridden by subclasses that need to
        implement this feature, e.g., Combinator and Operation.'''
        return self

    def __len__(self):
        '''Returns the number of elements available. This is closely
        related to '__getitem__'. If the node is not indexable, simply
        return zero.'''
        return 0

    def typify(self, type):
        '''Puts new constraints on the deduced type of the DataNode
        object.'''
        pass

    def substitute(self, new):
        '''Substitutes the data associated with this node with the
        data associated with another node. The substitution is a matter
        of redirecting the 'data' slot in this object's parent to a
        new object.'''
        self.parent.substitute(new)

    def __str__(self):
        '''Returns a string representation of the DataNode object.'''
        return self.__repr__()

    def error(self, format, *args):
        '''Prints an error message based on the source code location
        associated with the node.'''
        self.location.error(format, *args)

    def warning(self, level, format, *args):
        '''Prints a warning message based on the source code location
```

```
        associated with the node.'''
        self.location.warning(level, format, *args)
```

```
class Number(DataNode):
    '''Leaf node holding an immediate value, either of type integer
    or real.'''

    def __init__(self, value):
        '''Instantiates a new Number node - should be instantiated
        using Node(Number(#)).'''
        DataNode.__init__(self)
        self.value = value
        self.cached_type = Type.Int

    def type(self):
        '''Deduces the type of this particular DataNode instance.'''
        if type(self.value) is int:
            return self.update_type(Type.Int)
        else:
            return self.update_type(Type.Real)

    def typify(self, type):
        '''Puts new constraints on the deduced type of the DataNode
        object.'''
        if type == Type.Real:
            self.value = float(self.value)
        else:
            self.value = int(self.value)

    def __lt__(self, other):
        '''self < other <=> self.__lt__(other)'''
        return Node.compare(self, other, lambda o: o.value) < 0

    def __eq__(self, other):
        '''self == other <=> self.__eq__(other)'''
        return Node.compare(self, other, lambda o: o.value) == 0

    def __repr__(self):
        '''Returns a string representation of the Number object.'''
        return str(self.value)

    def __add__(self, other):
        '''x.__add__(y) <=> x + y'''
        return Number(self.value + other.value)
```

```
    def __sub__(self, other):
        '''x.__sub__(y) <==> x - y'''
        return Number(self.value - other.value)

    def __mul__(self, other):
        '''x.__mul__(y) <==> x * y'''
        return Number(self.value * other.value)

    def __div__(self, other):
        '''x.__div__(y) <==> x / y (floating-point number division)'''
        return Number(float(self.value) / float(other.value))

    def __or__(self, other):
        '''x.__or__(y) <==> x | y (integer division)'''
        return Number(int(self.value) / int(other.value))

    def __mod__(self, other):
        '''x.__mod__(y) <==> x % y (modulo)'''
        return Number(int(self.value) % int(other.value))

    def __invert__(self):
        '''x.__invert__() <==> ~x (logical negation)'''
        return Number(int(self.value == 0))
```

D.3.5 semantics/types.py

```
from options import Options

class Type:
    '''Representation of a type in the FCC language.'''
```

```
Type.Void       = Type(priority=0, name='0')
Type.Int        = Type(priority=1, name='Z')
Type.Real       = Type(priority=2, name='R',  is_real=True)
Type.IntList    = Type(priority=3, name='Z*', is_list=True)
Type.RealList   = Type(priority=4, name='R*', is_list=True, is_real=True)
Type.VoidFunc   = Type(priority=5, parameters=(Type.Void,),
                                   result=Type.Void)
```