Functional Modelling of Operating Systems

Doctoral Thesis

PÁLI Gábor János
Doctoral Student

Horia F. POP (UBB)
KOZSIK Tamás (ELTE)
Doctoral Advisors

EÖTVÖS LORÁND UNIVERSITY
DOCTORAL SCHOOL OF INFORMATICS

Doctoral Program: Foundations and Methodologies of Informatics
Head of School: BENCZÚR András, DSc.
Head of Program: DEMETROVICS János, DSc.

2012
# Contents

Introduction 3

1 Fundamental Problems 9
   1.1 Synopsis 9
   1.2 Abstraction Versus Performance 9
   1.3 Advantages of Higher Level Approaches 10
   1.4 Related Work 10
      1.4.1 Abstractions in Operating Systems 12

2 Employed Techniques 15
   2.1 Synopsis 15
   2.2 Language Embedding 15
   2.3 Building a Language Core 16
   2.4 Generic Optimizations in the Frontend 17
   2.5 Target-Specific Optimizations in the Backend 20
   2.6 Summary 22

3 Composing Little Languages into Applications 24
   3.1 Synopsis 25
   3.2 The Flow Language 25
   3.3 Global Configuration and Events 26
   3.4 The Flow Types 27
   3.5 The Flow Kernels 29
   3.6 The Flow Constructors 29
      3.6.1 Examples 32
   3.7 Abstract Programs 33
      3.7.1 Graph Decomposition 33
      3.7.2 Channels 34
      3.7.3 Tasks 35
      3.7.4 Mapping Flows to Programs 36
   3.8 Semantics 39
      3.8.1 State Transformers 41
   3.9 Summary 44

4 Real World Execution of Flows 46
   4.1 Synopsis 46
   4.2 Executing Tasks 47
   4.3 The Pool of Tasks 47
   4.4 Memory Management 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Scheduling</td>
<td>52</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Marking Boundaries for Kernels</td>
<td>52</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Selectors</td>
<td>52</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Creating Task Pools</td>
<td>54</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Example</td>
<td>56</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Parallelizing Abstract Programs</td>
<td>57</td>
</tr>
<tr>
<td>4.6</td>
<td>Code Generation</td>
<td>59</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Eliminating Redundancy</td>
<td>60</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Wrapping Nodes</td>
<td>61</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Interacting with the Embedding Environment</td>
<td>62</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Run-Time Support</td>
<td>63</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>Flow Programs as Operating Systems</td>
<td>66</td>
</tr>
<tr>
<td>5.1</td>
<td>Synopsis</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>Sketching Up a Dataflow</td>
<td>68</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Attaching <em>Feldspar</em> to the <em>Flow</em></td>
<td>69</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Anatomy of the High-Level Description</td>
<td>73</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Introducing Dynamism</td>
<td>74</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Marking Kernels</td>
<td>77</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Scheduling</td>
<td>77</td>
</tr>
<tr>
<td>5.4</td>
<td>Simulating the Dataflow</td>
<td>78</td>
</tr>
<tr>
<td>5.5</td>
<td>Compiling the Dataflow to C</td>
<td>79</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Added User Code</td>
<td>82</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Run-Time Support</td>
<td>85</td>
</tr>
<tr>
<td>5.5.3</td>
<td>The <code>main()</code> Function</td>
<td>87</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance</td>
<td>89</td>
</tr>
<tr>
<td>5.7</td>
<td>Summary</td>
<td>92</td>
</tr>
<tr>
<td>6</td>
<td>Related Work</td>
<td>94</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion and Future Work</td>
<td>98</td>
</tr>
<tr>
<td>7.1</td>
<td>Future Work</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>Acknowledgments</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>102</td>
</tr>
</tbody>
</table>
Introduction

Computers are part of our current lives and every one of them is made of two main components: hardware and software. Separation of these components is the result of an evolution where engineers have determined that it is more rewarding to manufacture hardware devices with a relatively low-level interface and then put additional virtual layers on the top of it formed by the programs run on it. As a result of the standardized structure of today’s computers, it is hard to find a machine without at least one layer of software associated to it, usually referred to as an “operating system”. With the spread of digital electronics, analog devices have been surpassed in favor of better maintainability and reliability of software-based digital devices.\(^1\) As a consequence, we are being surrounded by a plethora of embedded devices in the form of smartphones, routers, portable personal computers, multimedia sets, etc. that are still running some form of software on themselves.

Back in the early days of computing, having a computer was considered to be a privilege as there were only a few mainframes to serve all the requests coming from the terminals connected to them. Along with the change in the role of computers in our every-day life, the purpose of operating systems has also changed. Since the start of the new millenium, processor frequencies have plateaued, thus chip designers have had to scale processors in a new direction, namely the number of “cores” integrated on the same die. At the same time, there is the growing demand for building smaller yet reliable systems. It is obvious that the software methods applied in the previous decades have to be adapted to the new needs, which is a yet another challenge for software developers.

Despite the constant efforts of coming up with new ideas and approaches, operating systems’ development has continued to be based on the same fundamental technologies of the 1970s, when the UNIX operating system and the C programming language were first developed. Is this because everyone is satisfied with the results, because Dennis Ritchie and Ken Thompson found the philospher’s stone of computing? Well, as the history shows both of them continued to pursue the improvement of their own ideas (they eventually evolved into Plan 9, a famous research operating system), but they could not hold back their eldest child: it was licensed by enterprises and universities (e.g. University of California, Berkeley, the birthplace of the BSD UNIX), and thanks to the initial positive feedback it soon became the standard in operating systems.

Nowadays a serious obstacle in the spread of an operating system is lack of support for it. And without proper hardware support (or at least a proper documentation for the hardware) an operating system is virtually doomed to failure. However, there are certain segments of the market, where companies and software engineers are fed up with C and its derivatives, and they are seeking something that promises more automation in the development of low-level systems. The pace and direction of evolution of hardware

---

\(^1\)There are fields where analog electronics still performs better, though.
forces the programmers to start to think on a higher level, where the associated compiler
becomes their partner in the development of today’s applications. Such a typical field
of application of that methodology is tied to various domains, where more semantical
information can be extracted from the specification of the application to be constructed
for the compiler, therefore better target-language programs can be generated with less
human intervention.

The thesis features the following main contributions of ours as potential answers to
the challenges experienced in today’s industrial efforts.

– The Flow, a minimalistic glue language for combining programs written in different
embedded domain-specific programming languages. The purpose of the Flow is to
turn Haskell [33] into a specification language that is compact yet expressive enough
to describe complex applications as a composition of domain-specific-language pro-
grams.

– A high-level computation model based on dataflow networks that implements a
platform-independent way of representing complex applications. As a consequence
of the model, it is possible to specify a simple pure semantics for the application
(given that the contained components are themselves also pure), and it offers an
elegant way to control the execution of the application in a higher-level, declarative
manner.

– The design and implementation of a minimalistic run-time system for supporting
applications constructed with the Flow language. The structure of the run-time
system is strongly related to the computation model defined above, as its purpose
is to provide the same abstractions on every platform. Hence our goal was to
decrease the abstractions in the computation model to the minimum. It has several
advantages: it is easier to be ported to different hardware platforms and it is also
easier to give formal semantics for the elements of the run-time system.

The results presented in the thesis have been published or will be published in the
following research papers.

– Dévai, G., Gera, Z., Horváth, Z., Páli, G., Tejfel, M. Feldspar – A Functional
Embedded Language for DSP, Proceedings of the 8th International Conference on
pp. 149–156. [8]

Digital Signal Processing (DSP) algorithms are usually designed and described on
an abstract level and then transformed to a DSP chip-specific C code by expert
programmers. The problem is that the gap between the abstract description and
the platform-dependent code is huge and even the C code optimized for two dif-
f erent chips differs widely. This makes it expensive to rewrite the algorithms each
time a new chip is targeted. We expect that designing and implementing a high-
level domain-specific language (DSL) will make the implementation of algorithms
easier and a compiler together with platform-specific code generator and optimizer
modules will take the burden of target-dependent low-level programming off of the
programmers. To address these problems, we propose a new programming language

---

2So only the hardware vendor has to know its own device and therefore provide an optimizing compiler
or compiler backend for it.
called Feldspar (Functional Embedded Language for Digital Signal Processing and Parallelism). Feldspar features a compiler prototype which translates programs into hardware-independent ISO C99-compliant code, or recently, LLVM [40] intermediate representation.

This paper describes the basic concepts of contemporary language embedding in Haskell that is also influenced the design and implementation of our Flow language. In addition, Feldspar is used as prime example for a domain-specific language to be applied in conjunction with the Flow.


Software for digital signal processors (DSPs) is traditionally highly hardware-dependent and hence porting it to new processors usually requires significant design effort. In this paper we present Feldspar (Functional Embedded Language for DSP and Parallelism), an embedded, high-level, domain-specific language for DSP algorithm design and the compilation techniques we developed for generating C code from specifications written in Feldspar. While Feldspar allows description of algorithms on specification level, we show that with the right set of abstractions and transformations this high level, functional specification can be transformed into C code that is comparable or better than reference, hand-crafted C language implementations. The Feldspar compiler is highly modular and plugin-based, hence future hardware-specific plugins will enable automatic generation of efficient, hardware-specific code. This approach enables the encapsulation of knowledge of hardware completely in the compiler and thus allows description of algorithms in completely hardware-independent, portable manner.

This paper supports the thesis that embedded domain-specific languages can be indeed translated to efficient target-language (in this case, to C) programs. As the Flow language is built on techniques similar to the ones presented in the paper, it is also an evidence of our claims regarding the performance of the generated code.


A classic layout of complex software applications usually involves a set of fine-tuned performance-optimized routines that are combined and controlled from an upper layer in a lightweight fashion. As the applications grows, reliable operation, portability, and maintainability become a major concern. However, this can be tamed by abstracting away from the platform-dependent details by modelling the components and their relation on a higher level. Using a functional programming language combined with the technique of language embedding may be an answer to the question of how to implement such a solution. In this design, the component descriptions may be captured adequately by an embedded domain-specific language that compiles to a lower-level language but there also has to be mechanism for composition and thereby obtaining a complete working application out of them. In
this paper, a method for composing compiled embedded domain-specific languages into a stand-alone application with minimal effort is introduced.

This paper introduces the Flow language through a motivating example which is also featured in the thesis. It also describes the major elements in the resulting computation model, e.g. tasks, channels, and the execution of tasks. Though its focus is on the applicability of the approach in relation of arbitrary Haskell-embedded domain-specific language as that is one of the main concerns in the adaption among third-party developers.


As part of our previous work on the design and implementation of a coordination language framework for embedded domain-specific languages we briefly touched on the issue of executing specialized task graphs. In our model, we created a set of workers (executors) dynamically picked and executed DSL programs (tasks) as part of a dataflow graph. The number of workers matched the number of processing units of the given hardware in the ideal case. This last aspect may contribute to lowering the expectations from the supporting run-time environment, eventually making the compiled graphs standalone on top of the bare metal. However, during the preliminary performance tests, we observed that our first naïve stab at scheduling execution of graphs did not scale well for multiple workers – which is not surprising as relations between tasks induced by data dependencies in the dataflow graph are not respected. To provide guarantees for that, tasks are partitioned into pools, although the optimal organization of pools is hard to achieve automatically. Hence we decided to create another embedded domain-specific functional language to provide a way for the programmer to express heuristics on how dataflow graphs are scheduled, which lead us to the concept of “declarative scheduling”. Declarative scheduling allows the implementation of domain-specific scheduling constraints that helps to abstract away from the low-level scheduling details and rather focuses on the protocol implementation itself. Recent research work done on the topic in the context of databases and cloud computing supports the idea that gives us a motivation to evaluate the concept in our setting too.

Finally, this paper attempts to exploit the advantages of the Flow language, and proposes an extension to the model above. In that extension, a tuning for performance based on certain properties is enabled which shows how a practitioner may profit from describing the application on a higher level and leaving the details to an appropriate compiler.

In Chapter 1, we provide an overview of this area of research, a brief description of the problem together with an analysis of the related work. Our proposed solution is then gradually revealed through the chapters that follow.

In Chapter 2, we introduce the employed techniques that very specific to the field and approach of our work. There Haskell as a tool is used to define another programming language by the method called language embedding. The sophisticated type system and the flexibility offered by programming with functions makes Haskell quite suitable for working with language prototypes. As a working example of that, Feldspar, a domain-specific language for digital signal processing and parallelism is discussed briefly to illustrate the
concepts of embedding. Some parts of Haskell are already implemented with embedded languages (e.g. monads), thus Haskell strongly motivates the programmer to think in them.\(^3\)

In Chapter 3, a glue language for little languages called \textit{Flow} is described as our contribution. The goal is to maintain the same degree of composability is provided by Feldspar as described in the previous chapter, but on a higher level where programs are represented in various domain-specific languages. The purpose of \textit{Flow} is also to show how typical components in such a system may be captured in an abstract manner. Apart from that, semantics for an abstract machine is given to explain how to run the resulting programs.

In Chapter 4, we proceed with the analysis of programs in relation to efficiently running applications written in the \textit{Flow} language. The problems discussed include the way in which the components are scheduled for execution by distributing the available resources (processor and memory) among them. During the investigation, a code generation strategy together with a small run-time system is elaborated that contributes to minimizing the complexity cost resulting from the use of a functional programming language. As a plus, a user-controllable way of scheduling is defined that helps to parallelize the previously presented abstract machine.

In Chapter 5, the discussion continues on \textit{Flow} by taking an advanced example application to validate and demonstrate the design. However, our further intention with this application is to explicitly characterize and thus model a class of operating systems. Such systems are commonly deployed on embedded hardware, where the purpose of the operating system is to operate a hardware board dedicated to certain tasks limited to a specific application, e.g. digital signal processing. Hence we take Feldspar from Chapter 2 and create an extension for it to continue with programming such hardware on a higher level.

In Chapter 6, we compare our results with the current achievements of the field, discuss what the main differences are in our approach: what advantages and disadvantages it has in reflection to others’ results and answers to the related questions.

Finally, in Chapter 7, our conclusions on the topic are summarized in closing.

Note that we have chosen to use Haskell for formalization of the important results in the thesis. We believe that Haskell is abstract and strongly typed enough\(^4\) to provide a stable foundation for capturing and reasoning about the concepts we are discussing here. This approach coincides with the main goal of the thesis, that is, to model operating system components using a functional language. If we did not believe that such a functional model has advantages over the traditional way of engineering such systems, we would have continued to use pure mathematical models and would have presented them here instead.

According to the Curry-Howard correspondence, “a proof is a program, the formula it proves is a type for the program” (elaborated in [69]). That is, every program in Haskell is in essence a constructive proof of a formula represented by its type. Although we do not use the types directly as logical formulas in the document, when working on the results featured in this thesis, we also followed the same idea as we were trying to describe the given model as a program written in Haskell. Many of the attempts failed just because the types did not “add up” and thus lead us to re-think the design. Occasionally this approach

\(^3\)This may be confusing for the programmers in average, but a great win for the ones who realize the power of that.

\(^4\)If Haskell is used well.
identified several deficiencies. The Haskell type system and compiler proven to be very helpful in validating our ideas (both syntactically and semantically). Furthermore, they have proven to be invaluable tools in our research. It obviously took time to learn how to interpret the experienced results, and to think in terms of the types of the programs. However, we are now confident that “well-typed programs cannot go wrong”\textsuperscript{5} is indeed true. Let us note that, despite its high-level nature, Haskell works with low-level tools that directly follow mathematical concepts, sometimes reaching the realms of category theory. These tools provide the primitive building blocks for creating programs on top of mathematical abstractions, that is to say they implement a way of programming by mathematics. Features like parametric polymorphism, high-order functions, and type classes contribute substantially to finding and maintaining the essential elements of a solution to a given problem. Thus, although we try to introduce and explain every related concept to the reader, this thesis assumes some preliminary knowledge of functional programming. Types and programs are used to sum up the important points of the discussion and represent a formal counterpart for them.

\textsuperscript{5}From Robin Milner, the father of the ML language.
Chapter 1

Fundamental Problems

Modelling operating systems and various aspects of systems programming requires one to be familiar with the standard design and implementation of those ideas, and on the other hand, it also assumes advanced knowledge and experience in contemporary functional programming. The goal of the thesis is to discover and address performance and implementation problems experienced in the field, and contribute to the future development of operating systems written in functional, or other high-level programming languages. As we will see in the following, this is still an open question since high-level, especially functional programming languages are not yet wide-spread and are not currently adapted to everyday industrial use.

1.1 Synopsis

In this chapter, we briefly look through the possible reasons why a functional programming language would be hard to use for writing systems software, and what advantages it has over the standard techniques. It is then followed by a summary of many of the related works in the field to motivate and position our research presented in the thesis.

1.2 Abstraction Versus Performance

A potential barrier to the use of high-level languages in low-level and industrial application is the higher level of abstraction set they provide: that approach fundamentally excludes the support for low-level and platform-dependent parts since such support would risk both platform independence of the language primitives and consistency of abstractions offered by the given implementation. Unlike C++ for example, where one can work with elements defined in terms of higher levels while still employing the traditional C solutions for the parts requiring explicit knowledge of the actual implementation scenario. However, static type systems featured in modern functional programming languages are usually stronger and more strict than the one created for C++ \(^1\) therefore they simply do not allow the programmer to mix and match different weights of abstractions at his ease. This approach would be fruitful in the sense that strong typing facilitates and enforces writing correct programs or programs with reasonable semantics. It must also be noted that even C++ implementations work with a run-time machinery complex enough to not

\(^1\)Although the next upcoming C++ standard, C++0x ships with a strengthened type system, becoming more like the ones found in functional languages.
to be preferred as vehicles for operating systems development. Many popular modern and high-performance systems, like the Linux-based ones [70], or FreeBSD [51] are still developed in C.  

1.3 Advantages of Higher Level Approaches

Apart from performance, the security and reliability of operating systems are frequently considered. It is easy to write high-performance but unsecure and unreliable programs in low-level languages as many of the common programming tricks for achieving better performance stem from applying side-effects, and ignoring or breaking rules of typing. Consequently, introducing strongly typed high-level programming languages in systems programming is a great challenge indeed. However, in our opinion, performance costs of the more complex run-time support requirements may be compensated by the program correctness guarantees of static typing and the power of high-level abstractions. Both of these enhance both security and reliability at the same time. Programs written in high-level languages typically have fewer mistakes in their implementations. This is because the most of the error-prone part of the source code is derived, generated, or managed automatically in such environments, and because the syntactical constructs are usually closer to the exact program specification.  

1.4 Related Work

To summarize our current understanding of the application of functional programming concepts to operating systems in advance, it is very likely that operating systems may be developed using functional programming languages. The emphasis of discussion should be rather on the efficiency of finding a balance between performance and maintainability. Finding this balance is considered hard for embedded (resource-constrained) systems as most of the high-level programming languages, including functional ones, employ some form of automatic resource management, i.e. garbage collection. The use of garbage collection in resource-constrained environment is still an area of active research [21].

In addition to the problems that emerge from the implementation techniques offered by high-level languages, it is important to note that there are many operating systems that have been developed in research projects in high-level programming languages already and their source code is mostly open for study. In other words, our attempt is not the first in the field, the development of operating systems in high-level languages has been a topic of research for decades.

For long time, MINIX [68] was an operating system for educational and research purposes, but later it became a full-featured system in its own right, including compatibility with the POSIX standard, while its code base remained relatively compact and clean. It contains many enhancements in terms of abstractions over the concepts of the classical operating system design, and the highlights of its structure can be grasped, extracted, analyzed, and modified easily. Major enhancements include communication based on

---

2Partially because they have to maintain legacy code.
3It can be even observed in the implementation of high-performance Haskell libraries, they tend of hide them cleverly, though.
4Sometimes high-level programs are a form of the specification as high-level languages maintain a close relationship to mathematics and strong formalism making them very precise.
message passing between the system components in a strictly controlled manner, restriction and capability of on-the-fly resurrection of device drivers, a drastic reduction in the core functions over a traditional monolithic kernel. These are mainly the consequences of MINIX being a microkernel-based operating system. A microkernel is a kernel designed to be minimal both in code size and primitives. It provides the traditional advantages of the microkernel approach to system structure, namely improved reliability and flexibility.

L4 [37] is a second-generation microkernel, similar to MINIX mentioned above, and tries to overcome the performance limitations of the previous generation of microkernels. With implementation size on order of 10,000 lines of C++ and assembly code, it is an order of magnitude smaller than its ancestor, Mach [1] and two orders of magnitude smaller than its competitor, Linux. The correctness and reliability of any non-trivial system clearly critically depends on the operating system and its kernel. In terms of security, the operating system is part of the trusted computing base, i.e. the hardware and software necessary for the enforcement of the system’s security policy. It has been repeatedly demonstrated that current operating systems fail at correctness, reliability, and security. Microkernels address the problem by applying the principles of minimality and least privileges to the operating system architecture. To gain confidence in the overall system, it is therefore highly desirable to model or formally verify the correctness of this design and its implementation. As noted before, compared to other operating system kernels, L4 is very small – compared to the size of other formalization efforts, 10,000 lines of code is still considered a very large and complex system. There exists some work in the literature on the modelling of microkernels on the abstract level with varying degrees of completeness [37, 38, 39]. Bevier and Smith [5] specify legal Mach states and describe Mach system calls using temporal logic. Shapiro and Weber [66] give an operational semantics for EROS and prove a confinement security policy. Number of various case studies in the literature describe the IPC and scheduling sub-systems of microkernels in Promela and verify the formal descriptions with the Spin model checker [4]. These abstractions were not necessarily sound, having been manually constructed from the implementation, and so while useful for discovering concurrency bugs do not provide guarantees of correctness.

Faulty device drivers are a major source of operating system failures, and there are a number of static and run-time techniques have been proposed to detect and isolate such faults. Existing static analysis tools are capable of detecting a limited subset of errors, such as common operating system programming interface rule violations and certain memory allocation and synchronization errors. Stronger correctness properties, such as memory safety, race freedom, and correct use of device interfaces, are usually beyond the reach of these tools. However, run-time isolation architectures are capable of detecting broader classes of errors, at the cost of introducing a massive and therefore undesired overhead. An alternative to fault detection and isolation is an improved driver development process that guarantees correctness by construction. One way to achieve this is to synthesize device drivers automatically from a specification, thus reducing the impact of human error on driver reliability and potentially cutting down the costs as well. There is a tool, called Termite [64] which represents a concrete approach to driver synthesis by combining two formal specifications: one describing the registers and the behavior of the device, and one describing the interface between the driver and the operating system, to finally synthesize a complete driver implementation in C.
1.4.1 Abstractions in Operating Systems

Inferno [63] is an important cornerstone of the history of abstractions employed in operating systems. The roots of Inferno lie in Plan 9 [59], the intellectual successor to UNIX. Although Plan 9 featured many new interesting concepts and clever design choices compared to its predecessor (e.g. all resources are uniformly represented as files, construction and application of coherent namespaces, introduction of a standard communication protocol to access all resources independently of their actual location), it still has not gained the same popularity. However, Plan 9 clearly inspired the design and implementation of many modern operating systems or their specific components (e.g. Unicode). In Inferno, platform independence itself plays an important role in the design, and this intention serves as a foundation for capturing proper high-level abstractions for various operating system functions, studying them, and using them as a model. Thus it is not a coincidence that it has influenced several Java-based systems, for example because it enforces process limits by its own typed language, called Limbo, instead of relying on hardware support and by working with a special byte code directly developed for running programs.

Speaking of virtual machines in the world of operating systems, it is worth mentioning JX, [25] an operating system written in Java, which runs on an extended Java Virtual Machine. Most of the employed extensions cover functions those required for making certain operating system services accessible to applications. In this approach, different protection mechanisms provided by the given programming language become important for ensuring the stability of the system, thus establishing the concept of language-based system or language-based protection. In this setting, a compiler for the given programming language tries to check statically the same conditions (typically by employing the concept of types) in compile time as the hardware would do in run time [65]. By exploiting the assumptions behind the statically proven correct code, it is possible for such systems to balance the trade-offs introduced in their performance, or even exceed that of the traditional systems. We consider this direction especially important as language extensions and typing can be handled very well in functional languages.

Singularity [35] is based on the same ideas as JX is, but it is implemented on top of the Microsoft .NET platform. It introduces a protection model between processes implemented in software, called Software Isolated Processes that relies on a specialized compiler (Bartok). Singularity is mostly written in C#, the premier programming language of .NET, though for some components of the system different sub-languages are used for description, called Spec#, and Sing#.

There are some other additional approaches in the field, with functional concepts in their focus or they are already known to work with functional languages.

House [28] was originally developed for researches to explore the possibilities of functional programming languages, especially Haskell, in the area of systems programming. Since its initial version it has matured substantially and is still a subject of active research and development. Since it is primarily a research tool, it works with relatively simple elements, e.g. the \( \mathbb{H} \) monad for safe hardware access.

Hilde [73] also tries to build up operating system functions based on a functional language similarly to House. It is implemented in Clean, a lazy and pure functional programming language. It consists of two main components, called Fanke “kernel” and the Esther “shell”. In contrast to the standard interpretation, the former only implements communication between different processes over the network, and the latter aims to be a
tool for interacting with the user. This model addresses the solution in relatively elegant and clean way, because it ignores the low-level details as it relies on the run-time system of the Clean language and the operating system (currently Microsoft Windows) under it. Thus, we are only talking about a middleware that allows processes to cooperate in a distributed manner.

The Hello project [20] aims to design and implement an operating system in Standard ML [52] of New Jersey (SML/NJ), where an empirical study of the interaction between advanced programming languages and systems programming is given. The thesis enumerates the results of several projects and presents the design and implementation of the Hello operating system, including porting the run-time system of SML/NJ, constructing a kernel image for a bare personal computer, and writing SML software to access and manage the hardware and system resources.

The Glasgow Haskell Compiler has a rather sophisticated (and still rapidly developing) support for concurrency in its run-time system, which is written in low-level C code. However, as the compiler evolves, the run-time system becomes increasingly complex, more prone to human errors, difficult to maintain, and also difficult to add new concurrency features. Fortunately, there is an alternative approach to implement concurrency in GHC [43]. Rather than hardwiring all kinds of concurrency features, the run-time system is just a thin substrate that provides only a small set of concurrency primitives, and the remainder is implemented in Haskell libraries. This design improves the safety of concurrency features, which can be then developed as Haskell library packages and deployed modularly. A great deal of complexity is hidden inside the implementation of the concurrency abstractions. Much of this support takes the form of a run-time system that supports threads, primitives for thread communication (i.e. locks, condition variables, transactional memory), a scheduler, etc. A modern run-time system that supports running threads is clearly complex enough to be considered a miniature operating system, therefore solutions implemented in the terms of it may be adapted almost directly to the development of functional operating systems.\(^5\)

There is some recent work on showing how to extend a modern statically-typed functional programming language with features that make it suitable for solving problems that are common in systems programming [12]. Researchers are particularly interested in the problem of manipulating data with rigid representation requirements in a safe manner. Their work is based on the assumption that the constraints on the representation of the data are imposed by an external specification such as an application binary interface (ABI) or the data sheet for a hardware device. The design provides support for two classes of data types whose representation is under the control of the programmer. The first class consists of data types that are stored in bit fields and accessed as part of a single machine word. Standard examples of these can be found in operating system programming interface, in the control register formats that are used by device drivers, in multimedia and compression codecs, and in programs like assemblers and debuggers that work with machine code instruction encodings. The second class consists of data types that require a fixed representation in regions of memory. Often those “memory areas” are specific to a particular processor architecture, hardware device, or operating system kernel. Diatchki’s approach builds upon well-established programming language technology, such as type inference, polymorphism, qualified types, and the use of monads to control the scope of effects.

Mirage [47] is the result of promising research that focuses on the domain of cloud

\(^5\)As we have done in our research as well but with a different set of abstractions.
computing, and features a specialized software stack for achieving that goal. Mirage is structured as a “vertical operating system” a’la an exokernel. The key principle there is to simply abstract away from the hardware details beneath, by delegating the task to an operating system hypervisor, e.g. Xen [3] or Hyper-V, and work with a virtual hardware instead of a compiler target for a high-level language. Hence Mirage is capable of hiding the complexity of programming against the standard but very low-level interface of such hypervisors by incorporating a suitable compiler toolchain that gives a programmer high-level abstractions. Currently, their prototype implementation uses the OCaml language [62] as they previously managed to construct high-performance applications using this language. In addition, they did not have to modify the OCaml compiler itself, only the run-time libraries were adjusted to provide an interface to the operating system.

The Haskell Lightweight Virtual Machine (HaLVM) [22] appears to implement a similar solution. It is a port of the Glasgow Haskell Compiler’s run-time system for Haskell to run directly on Xen. Because of that, there is no barrier between the kernel and the userland that makes writing low-level code efficient, while the applications can be written in Haskell.

With the advent of multi-core processors, and the growing need for implementing cloud computing services [72], the concept of factored operating systems [74] – created by the developers of the Tilera processor family – is also closer to being realized. Here, the design implements scheduling based on space sharing instead of time sharing to increase scalability. Each operating system service is factored into a set of communicating servers which implement the service in question together. The servers are bound to distinct processing cores to avoid competing for implicit resources (e.g. caches) with end-user applications. Services are strictly isolated and they can only communicate with other services and applications via messages. There may also be a level of fault tolerance implemented in this manner, as requests to the services may be re-routed between the set of cores implementing it. This design comes with many advantages for processors with many cores: the operating-system-level communication is made explicit and exposed, thus there no need to laboriously identify performance bottlenecks result from the use of shared memory or locking; the number of servers to implement services scales well with the number of cores; execution of system and application code can be forced to be run on different cores at all times so expensive context switching can be elided.

Finally, Barrelfish is a multi-kernel operating system [67] that was conceived as a research project to explore possibilities in structuring operating systems for today’s and the future’s hardware. Barrelfish features a development framework, named Filet-of-Fish to address the aforementioned challenges. In this framework, the authors have chosen to embed C into a functional language, Haskell. Thus they replaced the pure concatenation-based core generation with safe and correct composition of higher level building blocks, resulting in a more bug-free compilation. Filet-of-Fish gives strong static guarantees in that the generated code is valid by construction and it can be always compiled.
Chapter 2
Employed Techniques

In this chapter we present the techniques which we employed to achieve our goal: as we will show in this chapter, we are trying to use Haskell as a tool for describing other programming languages. It is not uncommon for Haskell at all as it greatly supports and encourages formulation of smaller, restricted languages on top of its basic constructs. Haskell itself is built up in a similar fashion – that is not accidental too: in the following sections, we will give examples of why such a structure is favorable in the definition of languages or even any complex application programming interface.

2.1 Synopsis

First we introduce the concept of language embedding that presents the reader with how Haskell may be bent, restricted, or extended according to the current needs which then may be considered a different language. The resulting programming (or specification) language inherits many of the properties of the embedding, i.e. the host language, like the function-oriented nature or purity in case of Haskell. Maintaining purity is especially important from the side of reasoning and doing so will give us an elegant way to describe the connected semantics in the later chapters. It is followed by the complementary technique of building up a language by stacking layers of syntactic sugar which may make the process extremely flexible both for the developers and the users.

Besides syntax, semantics also has to be defined for the language in question. As it will be shown through a working but simple example, semantics may be given as an interpreter that maps the basic language constructs to regular Haskell functions or as a compiler that translates them to another language. The compiler solution discussed here follows the standard “frontend-backend” separation where the frontend is to implement generic optimizations, and the backend is to do target-specific optimizations. As we will see, the frontend communicate to the backend through an intermediate representation that facilitates the use of different backends with the same frontend therefore making it modular.

2.2 Language Embedding

Designing and implementing a new strongly typed programming language from scratch is challenging and generally takes a great deal of time. Nowadays one might find clever
lexer and parser generators for many popular languages but the same cannot be told for type checkers, thus it is still a very experimental topic.

To overcome the difficulties introduced by the lack of those “type checker generators”, language embedding has become an option. With this technique, a new language is implemented by using an existing one, called the host language. As a consequence, there is no need even for a lexer, parser, or a type checker as those tasks are performed by the corresponding components of the host’s compiler. In this case, a new language can be defined as a library consisting of functions not implementing actual computations but resulting in a data structure which is virtually an abstract syntax tree of the embedded program. Other functions can then apply different interpretations to the generated program, for example, evaluate it, or compile it to a given target language. An implementation based on such an intermediate structure is usually referred to as deep embedding. This technique enables various compile-time optimizations in a very natural way and offers quick language prototyping as well.

Haskell is a perfect choice as a host language for embedding, it is easy to work with such data structures in that language because it abstracts away from the details of memory management (e.g. pointers, allocation and release of memory regions, etc.), and features many sophisticated compiler optimizations connected to the high-level description of data structures, called algebraic data types. Static typing and functional nature of the language may be also inherited from there.

2.3 Building a Language Core

For the ideal formulation, it would be beneficial to find a minimal set of language constructs that are able to express a more verbose interface. It is commonly employed in many programming languages, even in Haskell itself. There is a so-called “core” language that contains the elements of the language base. This what the language backend (e.g. the code generator) usually receives and uses, it is the lowest level of building blocks for the language frontend. The actual language itself is built above that core by adding several types of “syntactic sugar” for the constructs within the core. The purpose of this design choice is to create a small set of basic blocks that can even be described formally and implemented more easily.

The core language is a complete programming language alone, but it is usually too simple and raw to be exposed directly to the user. That is why such languages are “sugared”: they require additional layers that make it more acceptable for daily use and support concepts more relevant to the given application domain. And because the interface represented by those external layers should be permanent and stable, the designer has the opportunity, albeit a limited one, to change the core language without causing major disruptions in the upper layers. This approach is considered useful when one has to add new user-level constructs for the user interface because that may not require any major change within the language itself.

Because the so-defined core language technically results in a data structure, it can be interpreted, i.e. we can implement a simulator for the given system to study its operation without generating code. Through the interpreter we can also give formal semantics for each of the base constructs as writing programs in Haskell is comparable method to providing a precise mathematical description.

To demonstrate the concepts and introduce a contemporary embedded language in
sumSquares :: Data Int -> Data Int
sumSquares n = (sum . map square) (1 ... n)
  where square x = x * x

Figure 2.1: Feldspar program sumSquares

Haskell for later use, we are going to mention Feldspar as an example here.

2.4 Generic Optimizations in the Frontend

Implementation of Feldspar provides two kind of libraries: the core language, and its high-level interfaces, representing different features towards the user. Programmers need to import these Haskell libraries and write Haskell programs which use their exported functions in order to write Feldspar programs. Thus Feldspar program sources are Haskell program sources but they start with the following prologue.

import qualified Prelude
import Feldspar

That instructs the compiler to hide the basic Haskell functions – they remain available later, but they have to be qualified for access, while importing the basic Feldspar functions. That is, the semantics for the current program source may be plugged in or replaced completely this way. The Feldspar module here exports all the elements of its core language.

The core language introduces high-level imperative-like constructs (for instance, loops) with pure functional semantics, while the libraries implemented on top of this establish a specific application programming interface. For example, one of those libraries is called the vector library which reimplements some of the standard Haskell list functions to be able to transform them to a core language program. Since vectors are elementary to Feldspar, we will use them as an example for detailing the implementation of a high-level library.

In the vector library (the Feldspar.Vector module), the high-level library interface visible for the user is mainly based on the Vector data type. Vectors can be manipulated by functions similar to list operations in Haskell. Vectors can be also treated as subscripted sequences, allowing programs to be written as if they were presented in purely mathematical manner. Moreover, this formulation encourages formal reasoning about the programs, making them easier to analyze, for example.

An example of using vectors for representing formulas would be the calculation of sum of squares between 1 and n. Figure 2.1 shows a Feldspar program (sumSquares) implementing that with a list-like structure. The first line is a type signature that declares the function that both accepts and returns an integer-value argument. The right-hand side of the definition should be read from right to left: it starts by forming a vector containing elements from 1 to n then each element is mapped to its square using map then finally the elements of the squared vector are summed. Note that sumSquares is written as a composition of simple functions, which is typical in functional programs.

Though sumSquares seems to be a program that works with integer data, it is actually a program generator that produces another program upon its evaluation which will
compute the specified function. Programs generated this way are formulated by the core
language which serves as an interface to potential backends. In that sense, the actual
programs refer to construction of abstract syntax trees rather than the computation it-
self. Then those trees are received, transformed, and translated further by the specific
backend implementations.

The types use Data Int instead of Int – in general, Data a is the type of Feldspar
program computing a value of type a. The user can evaluate a program at the prompt of
the Haskell interpreter.

*Example> eval (sumSquares 10)
385

A function called printCore can be used to show the core language code is generated
from sumSquares, see Figure 2.2.

Given suitable helper definitions, the produced code may be run as if it were an ordi-
nary Haskell program. It contains a number of simple variable definitions and a single use
of the while function, corresponding to a C-style while loop. It is the absence of hidden
state in the core language that makes it purely functional. Instead, the while function
operates on an explicit state which is initially (0,0) and finally bound to (v11_0,v11_1).
It can be seen that there is only one while loop, despite the fact that the function was
composed by three distinct vector operations: building up the vector, mapping the el-
ements to their squares and finally summing up the result. If those operations were
compiled separately, they would result in a separate loop each.

The use of vectors in sumSquares may also raise worries of excessive memory con-
sumption of the resulting program. However, all by these can be avoided since all the
vector operations can be “fused” together [24]. This level of efficiency is obtained by the
removal of intermediate data structures, a technique known as deforestation or fusion.
Fusion techniques enable the implementation to compute vector elements on demand,
Figure 2.3: Simplified implementation of a Vector.

\[
\text{map } f \ (\text{Indexed } l \ ixf) = \text{Indexed } l \ (f \ . \ ixf)
\]

Figure 2.4: Working with a Vector.

and source programs are mostly simplified at compile time even when the abstract syntax tree to be translated is under construction. Most functions technically rewrite the input program at a macro level. There are several stages at which a Feldspar program gets rewritten.

To illustrate the process of program rewriting we take vectors. Vector operations may be defined by manipulating the length and the index function only. According to this observation, we can give the definition of the type \texttt{Vector} as it can be seen in Figure 2.3. Note that the representation introduced here differs from one is actually implemented in the current version of Feldspar for the sake of simplicity.

A vector in this context a record consisting a length and an index function that maps each index to an element in the vector. Thus vectors do not appear in the memory in their full lengths. Let us take the definition \texttt{map} as an example to show how to work with such representations. The second parameter is a vector, where we are matching a pattern, i.e. decompose the contents of the passed value. According to the result of the pattern matching, \( l \) will hold the length, and \( ixf \) will be the index function of the given vector.

Applying \texttt{map} to an arbitrary \( f \) function and \( v \) vector results in a new vector of the same length as \( v \) but with a different index function. The new index function is simply a function composition of \( f \) and the old index function.

Representing vectors by their index function has the benefit of allowing a mathematical style of programming. For instance, a mathematical definition of \texttt{map square a} might be given as it is shown below.

\[
b_i = a_i \times a_i \quad \text{where } i \in \{0, N - 1\}
\]

The translation of this formula to Feldspar is most trivial.

\[
b = \text{Indexed } (\text{length } a) \ (\lambda i \to (a ! i) \times (a ! i))
\]

The \texttt{sum} function can be written using the corresponding core language construct, a \texttt{for} loop to the core language.

\[
\text{sum } (\text{Index } l \ ixf) = \text{for } 0 \ (l - 1) 0 \ (\lambda i s \to s + ixf i)
\]
Arguments of for are as follows: a start index, an end index, and a start state respectively followed by a body as an anonymous function parameterized over the actual index and the current state accumulated over the loop. A vector element at index \( i \) is added to the running sum in each iteration.

The \( 1 \ldots n \) expression can be viewed as a function generating a Vector with \( \ldots \) as an infix operator.

\[
1 \ldots n = \text{Indexed} \; n \; (\lambda i \rightarrow i + 1)
\]

By combining map with definition of square and \( 1 \ldots n \), the high-level compiler builds a simplified program out of the sumSquare function. As a result, vectors will completely disappear there and their index functions will be merged into the body of the for loop.

\[
\text{sumSquares} \; n = \text{for} \; 1 \; n \; 1 \; (\lambda i \; s \rightarrow s + (i \; \times \; i))
\]

It is worth noting that it is possible to avoid recomputation of elements in a vector that are accessed multiple times. In order to do so, there is a special identity function, called memorize provided that writes all elements to memory – in contrast to other vector operations, it explicitly allocates memory for the vector.

Not all language constructs need to be part of the core language. Using the host language to generate programs allows us to define complex language constructs as generators that translate into more primitive constructs. For example, the for loop in Feldspar is internally expressed as a while loop. Thus, the user has access to both kinds of loops, but the backends only need to support one of them. This is very similar to the way that most high-level languages use syntactic sugar to translate certain syntactic formulations into primitive ones.

As we have shown in this section, this approach has been taken quite far in Feldspar. While the user has full access to the core language, a typical high-level Feldspar program is mostly expressed using smart generators that can even perform high-level optimizations on the fly. Thus it is common to build a domain-specific language around a simple core language and a set of functions or operations specific to the given problem domain. This gives the user the feel of working in a much richer language.

### 2.5 Target-Specific Optimizations in the Backend

In general, target language backends produce platform-dependent code from the intermediate representation that core programs represent on a higher level. Here we present a backend for generating C programs but it is only one of the possible ones. A compiler structure including backend used for Feldspar is shown in Figure 2.5 where programs are compiled further to C with an input of a core language program which is represented as a graph. There is a generalized algebraic data type used to represent this graph which allows the constructors to be completely type-safe with regards to the values flowing in it.

The core graph is first transformed into an abstract imperative code. Note that the abstract imperative code is no longer purely functional. The transformation is straightforward as local variable declarations, primitive operations, branches, and loops are represented as nodes in the input graph and they can be mapped directly to the abstract
imperative counterparts. Initialization and update of loop states are also taken care of there. Types used in the abstract imperative code is different from that of C. A range of fixed-size signed and unsigned integer types is used, arrays with fixed length, floating-point, and Boolean types. Mapping these types to the corresponding C types heavily depends on the given platform.

In C, pointers and operations on pointers are used to handle parameter passing and arrays as there only values can be passed. That makes implementation of optimizations harder because pointers have to be tracked through the program for their correct use. Hence the abstract imperative code hides pointer operations: for each parameter and local variable its logical type and role (input, output, or local) are stored telling us how parameters of variable should appear in certain positions within the resulting C code.

```c
#include "feldspar_c99.h"
#include <stdint.h>
#include <math.h>

void sumSquares( int32_t n, int32_t * out ) {
    int32_t var11_0;

    var11_0 = 0;
    (* out) = 0;
    {
        while (((var11_0 <= (((n - 1) + 1) - 1)))) {
            int32_t var6_0;
            int32_t var8;

            var6_0 = var11_0;
            var11_0 = (var6_0 + 1);
            var8 = (var6_0 + 1);
            (* out) = (((* out) + (var8 * var8));
        }
    }
}
```

Figure 2.6: Final, optimized C version of `sumSquares`.

The result of the first pass is a naïve program which might be subject to several optimization steps. It is a direct translation of a core language program, so it includes
many variables from the abstract imperative code. Some of them are the result of various program transformation steps, and the C transcript uses a few additional variables to properly implement the semantics. For example, most of the redundancies can be eliminated by copy propagation. Copy propagation is a well-known solution for replacing occurrences of variables with expressions assigned to them. However, this may have negative impact on the resulted code when multiple occurrences of a variable are replaced with a larger expression. The backend tries to avoid this by using the following heuristics: a variable is eliminated if either the expression assigned to it is relatively cheap to compute or there is only one occurrence to be replaced. Backward propagation is also supported to eliminate copying of arrays. An optimized version of the naive translation is shown in Figure 2.6.

A number of different hardware-dependent optimizations may be applied as well. These algorithmic transformations are part of the hardware-specific backend of the compiler, thus they may vary between different types of hardware. There are a number of common tricks which can be used generally. The compiler is designed to do the same optimizations automatically as a programmer would do when porting the given algorithm to a new platform. For example, there is loop unrolling which is a trade-off between speed and code size. The optima for loop unrolling depends on hardware, thus it is usually handled by hardware-dependent manual tweaks. It is also important for the compiler to gain extra performance since it is able to exploit the on-chip parallelization offered by most processors when number of internal execution units is considered when unrolling loops. The transformed code can be refined further by inserting additional keywords at the right places that will guide the C compiler that will emit better code. An example of such keyword is \texttt{restrict} that assures the compiler that the memory referenced by an annotated pointer will not have any aliases. Instructions in an unrolled loop body may be reordered and grouped in the future in order to be replaced with hardware-specific intrinsics, which would be a key in boosting performance to a level matching with hand-written C programs.

Optimizations are implemented in a highly modular way as plugins of a framework. Each plugin consists of an analysis and a transformation routine. Analysis determines whether the given transformation is valid to specific parts of the program, and collects semantical information. Different types of semantical information can be attached to the program nodes in a polymorphic fashion. After all the information required for the transformation is collected, the transformation has to traverse the program’s abstract syntax tree. During this traversal, it performs a given computation at each node. The plugin-specific functions have to be implemented. This approach provides for easier creation and maintenance of the plugins. When all the optimization steps are run, hardware-independent valid C99 code is produced via a pretty printer.

2.6 Summary

As a preparation for the presentation of the main contributions of the thesis, we have given a short introduction to the programming techniques we employed to use Haskell as a specification language. The interested reader may find more information on language embedding in \cite{14} and \cite{19}. As it was noted at the beginning of the chapter, the concepts are explained through examples because the method of expressing an object language in terms of a meta language is a difficult subject to present and elaborate formally, and that
is not part of our research efforts. Thus the chapter is rather to give an insight into the main ideas that we have employed in our research work.

In our opinion based on preliminary experience with embedded languages, it is an extremely flexible tool that helps to abstract away from the actual code generation since many different target languages may be supported by implementing the respective backends. Haskell has an extensive and rich ecosystem that freed us from reimplementing many of the tools required for the research. The approach also has the advantage that executable semantics may be described for each of the constructs to be used as a reference when implementing code generators. As we will see later in the succeeding chapters, the meta language also provides a formal framework to express relations between the elements of the object language. Thus making it possible to model them and formalize their logical connections while there remains the possibility of generating code for the constructed model optionally which may even compete with its hand-written counterpart in terms of performance.
Chapter 3

Composing Little Languages into Applications

In order to build functional models for operating systems, we start with the definition of a modelling language on top of a functional programming language. Because of this relationship, the modelling language inherits many of the features of the host that make it similarly functional. That inherited functional nature comes with several added benefits that we can exploit to approach problems in different areas in development of operating systems.

For example, composability is considered globally important for the entire discussion. Our proposal is to express operating systems as programs represented by dataflow networks, where the complete systems is a single function that maps input values to output values. The dataflow network is used to give the definition of that function as a composition of smaller, sometimes primitive functions. It enables frequently used components (as functions, of course) to be re-used during the composition that can also build up a standard set of primitives suited for constructing such systems. Another special property of functional programming is that functions are first-class values, i.e. functions may both take functions as parameters and return functions as results. This property leads to the concept of programming with higher-order functions that is a commonly applied solution to capture function templates, adding another (rather powerful) type of patterns to be recorded.

In addition to that, language embedding encourages the implementation each of the components in a “little language” that may be better suited to the given application or to use such existing languages to construct systems to be modelled. Since the meta (and host) language remains the same for every object language used in the description, they will not make a difference in the overall reasoning as long as they remain pure in their semantics. The little languages also help to raise the level of the description as they may include restricted constructs specific only to a certain domain, so the code generator has more opportunities to understand the underlying semantics better and therefore generate better programs, eventually contributing to efficient program generation based on the model. That is why it is so important to establish a methodology on how to compose domain-specific languages embedded in Haskell into larger applications.
3.1 Synopsis

In this chapter, we are going to use three different layers of languages: the meta language, Haskell, which implements a glue language Flow, which coordinates domain-specific-language programs with a specified interface. The Flow language is defined to capture schemes referred above and to provide a way to connect domain-specific-language programs into a dataflow network.

We give the basic types, elements and constructors of that combinator language, followed by their semantics as abstract programs, while touching the generic interface for fitting domain-specific languages to it. As we will see below, dataflow networks are first represented by an algebraic data type with little programs in its nodes which then mapped to a mathematical graph. Through the decomposition of the graph by its edges, we reorganize the relevant parts to channels and tasks that are going to be the key concepts in explaining the semantics for the Flow programs. In parallel with that, we will have to discuss how much dynamism is allowed by the language. It is a valid claim that we shall be able to describe changes in the behavior of the system over time. While the general structure of the Flow programs is static as nodes cannot be moved, created, or destroyed in run time, it is possible to control them by variable parameters. We may attach certain parameters to the nodes that may help in changing the behavior of the individual nodes (e.g. enabling or disabling them, routing data between nodes, etc.) dynamically.

3.2 The Flow Language

The goal of the framework to be introduced here is to identify and collect the basic operations needed for constructing dataflow networks. Hence its name, Flow. It is implemented as a hierarchy of modules in Haskell. The constructor operations help to describe a data dependency graph of nodes for the programmer. There can be various computations placed in the nodes. These computations can be described by programs written a domain-specific language. Such computations may have parameters. We call those graphs flows.

An example for a flow is shown in Figure 3.1. The flow itself always stands for a loop in an event-driven system, where there are source nodes to receive input and sink nodes to produce output. Source nodes are the nodes that do not have any preceding node (parent) in the graph. Sink nodes are the nodes that do not have any succeeding node (child) in the graph. As it can be seen, both of them represent side nodes. Contents of the graph are
executed continuously (and possibly in parallel, see later): source nodes incrementally
read values to input and the sink nodes incrementally write values to output, and that
is repeated forever.

The usual way in which such flow networks work is as follows. There is a set of
programs implemented in a domain-specific language. Then the programmer takes these
programs and connects them into a graph by specifying their order of dependency. It is
his responsibility to correctly compile such a graph for solving the problem in question.
But connections between nodes – the edges – are typed, so only nodes with matching
types can be connected.

The so-constructed graph can be both run directly in an interpreter of a Haskell
implementation (e.g. GHCi for the Glasgow Haskell Compiler) or compiled to a program
of a target language (e.g. C). In order to work with the flow, we shall provide input and
a set of parameters to the programs in the graph in some format.

– When interpreted, the input and program parameters must be specified as infinite
lists that the flow can consume during execution.

– When compiled, the real source of input depends on the client implementation. The
client is part of the target-language sources that is not generated from the flow but
manually added to make it compile. It is typically the implementation of the source
and sink nodes. In addition to that, the client must contain a function that may
set the program parameters.

3.3 Global Configuration and Events

As we have seen above, in order to use programs as components for a system to be
constructed, they must be wrapped as nodes. Other than wrapping, one may assign
parameters to the given program. A set of run-time parameters associated with all the
programs in the dataflow configuration is called the global configuration. Such parameters
may be updated when the system is running, and programs will receive the updated
parameters on their next run.

We wrap programs (as functions) only with a signature as follows. The $f$ is the original
program written as a function with type $a$ that refers to the input type, and type $b$
that refers to the output type of the program.

$$f :: c \rightarrow a \rightarrow b$$

Type $c$ is typically an n-tuple that contains all the run-time parameters for the whole
system, that is why it is called global. This helps to eventually find the intersection of sets
of parameters between functions. It may be considered as a simple form of communication
for coordination, where the unneeded parameters may then be masked. An initial value
for the configuration can be specified that may be changed while the system is being
executed. For illustration, see Figure 3.2. A detailed example will be given in Section 5.4.

This system-wide set of parameters is read-only for the internal elements of the con-
structed system. It is because we would like to keep a reliable reasoning behind the model.
Thus the global configuration can be taken as another parameter for each element to be
executed. The global configuration can be only changed from the outside – this is what
the client may access and modify at run time. For that case, having all the parameters
for all the programs in the network contributes to the ability to identify each piece of the configuration properly. That is, with this formulation, when the configuration is translated to the target language, the client will have the same elements appearing in the same order as they can be seen in the high-level description.

The representation of program parameters introduced above implicitly enables the environment to compose events. Events may be taken as changes in a subset of parameters in the configuration that cause the dataflow graph to change its behavior, that is, it will return a different output for the same input starting from that moment after the event got processed.

### 3.4 The Flow Types

Let us introduce the elements of the Flow language now. There is a common data type, FlowType, employed in the graph to connect domain-specific-language programs. That type must be able to connect and support communication between programs written in different little languages; thus it must be independent any of them – and the languages used in the network do not know about each other, too. Hence such types are represented by a restricted set of conventional Haskell types that provide instances for the following type class.

```haskell
class FlowType a where
  toFlowType :: a -> Chunk
  fromFlowType :: Chunk -> a
  flowType :: a -> FlowT
```

The functions in the class guarantee that the given type may be translated to the Chunk that is used to encapsulate data travelling between nodes. It is also important for the possible backends, because those are the types that they must implement for code generation. The Chunk type can be given as follows.

```haskell
data Chunk = CHBool Bool | CHInt Int | ...  
```

Note that the flowType function may be used to get type information about the Flow types in compile time, which is exploited in the process of code generation.

```haskell
data FlowT = FT_Bool | FT_Int | ...  
```
Each language to be used in the dataflow network – denoted by \( l \) below – has to provide a mapping for all of its own types. The \texttt{DSLType} (as “domain-specific type”) type class is used to describe such mapping between the types of the DSL and the \texttt{Flow} indexed with the given language.

\begin{verbatim}
class FlowType (T l a) => DSLType l a where
type T l a
fromDSL :: l -> a -> T l a
toDSL :: l -> T l a -> a
\end{verbatim}

When running a dataflow network, the functions \texttt{fromDSL} and \texttt{toDSL} implement the conversion between the data type \( a \) and the language \( l \) and \texttt{Chunk}. The type \( T l a \) corresponds to the \texttt{Flow} type associated to the DSL type.

For example, a sample instance can be given for a type that is used in one of the languages as follows. The purpose of that type is to represent the concept of the associated language. For \texttt{Feldspar} in Chapter 2, we used the \texttt{Data} type constructor to tag all the supported types there, e.g. \texttt{Data Int} for \texttt{Feldspar}-enabled integers.

\begin{verbatim}
instance DSLType Feldspar (Data Int) where
type T Feldspar (Data Int) = Int
fromDSL Feldspar x =
  case ((dataRep . eval) x) of
    IntData v -> fromIntegral v

toDSL Feldspar = value . fromIntegral
\end{verbatim}

The first line is straightforward: we tell the compiler that in the case of \texttt{Data Int} the type \( T l a \) in the signature of the \texttt{fromDSL} and \texttt{toDSL} function is replaced with \texttt{Int} (which satisfies the constraint of being the member of the \texttt{FlowType} class). According to this, the signature for the members of the instance will be as follows.

\begin{verbatim}
fromDSL :: Data Int -> Int
toDSL :: Int -> Data Int
\end{verbatim}

Thus the function bodies given in the instance definition above will implement the conversion between a \texttt{Feldspar} integer and a Haskell integer. In the definition of \texttt{fromDSL}, we have extract the represented integer value so we evaluate it by calling \texttt{Feldspar’s eval} function then we ask for the internal representation of the result by calling \texttt{dataRep}. That function then returns a value that contains information on both the type and the value of the argument, so they can be checked and extracted by pattern matching on it. In the definition of \texttt{toDSL}, the situation is simpler, because the \texttt{value} function of \texttt{Feldspar} is polymorphic so it can directly map the value to a \texttt{Data Int}.

We can also see that the \( l \) is now defined in the following way.

\begin{verbatim}
data Feldspar = Feldspar
\end{verbatim}

The example will be concluded further later in Subsection 5.3.1.
3.5 The Flow Kernels

The next layer of building up a language binding is the definition of an instance of the PrimKern type class. The PrimKern class briefly summarizes all the requirements towards the language to be used in the Flow program. It is because there are certain restrictions to make the code generation for the Flow-language models efficient. We will not detail those restrictions in this chapter, they will be discussed in Section 4.6 as part of the description of the code generation. For now, only the run will be mentioned, see below. Note that, besides the source (l) and the target language (t), there is also a k type variable. The k data type (as in “kernel”) is used to tag the domain-specific-language program in question with the values of l and t for later processing.

```haskell
class Backend t => PrimKern k l t where
    run :: (DSLType l a, DSLType l b, DSLType l c) => k a b c l t -> T l c -> T l a -> T l b
    -- for code generation only
    compile :: k a b c l t -> ID -> (Types t, Name, Body t)
    finalize :: k a b c l t -> Name -> Body t -> Definition t
    heapInfo :: k a b c l t -> Types t
```

PrimKerns may be created from programs of language l of type \( \gamma \rightarrow \alpha \rightarrow \beta \) if l has an instance for the Liftable type class.

```haskell
class Backend t => Liftable l a b c t where
    liftPK :: (DSLType l a, DSLType l b, DSLType l c) => l -> ID -> (c -> a -> b) -> PrimFlow (T l a) (T l b) (T l c) t
```

The liftPK function is employed inside a generic lift operation where a unique identifier (ID) is assigned to each lifted program. As we will see later, lifting essentially corresponds to wrapping the DSL programs as nodes in the graph, where PrimFlow is such a node. In order to be able to simulate a flow there must be a way to simulate the kernels themselves. Therefore it is required to have a function to run them, i.e. map them to regular Haskell functions. Though the details of running a Flow will be discussed as part of the semantics in Section 3.8), we can say in nutshell that “running” means evaluating a Flow program with some configuration (see Section 3.3) and input parameters to learn what output it returns.

We will return to the role of the Backend type class and the other functions in the PrimKern type class later in Section 4.6.

3.6 The Flow Constructors

A Flow is represented by a Generalized Algebraic Data Type (GADT for short) [58]. It contains all the primitives that can be used for constructing flows. We use a GADT to specify the language with deep embedding.

```haskell
data PrimFlow :: * -> * -> * -> * -> * where
    Kern :: (Backend t, PrimKern k l t, DSLType l a, DSLType l b, DSLType l c) => ID -> k a b c l t |
        -> PrimFlow (T l a) (T l b) (T l c) t
```
Seq :: (Backend t, FlowType a, FlowType b, FlowType c) => PrimFlow a b c d t -> PrimFlow c b d t -> PrimFlow a b d t
Fork :: (Backend t, FlowType a, FlowType b1, FlowType b2, FlowType c1, FlowType c2, FlowType d) => PrimFlow a (c1,c2) d t -> PrimFlow c1 b1 d t -> PrimFlow c2 b2 d t -> PrimFlow a (b1,b2) d t
Skip :: (Backend t, FlowType a, FlowType c) => PrimFlow a a c t

The heart of the data type is called PrimFlow that has five parameters (it is denoted by the five star symbols above which is the kind of the data type – that is, PrimFlow is a five-parameter type constructor, not a concrete type). However, in order to talk about parameters for the Flow type itself, we must resolve some additional details first.

type ID = Int
type Labelled a = State ID a
type Flow a b c t = Labelled (PrimFlow a b c t)

The ID is used to identify nodes in the network. Each node has a unique identifier that is used during its compilation. These identifiers are generated automatically by using a State monad (the Labelled type) that wraps PrimFlow and gives us the final type of Flow. The type parameters (a, b, c, t) for Flow refer to the input, output, global configuration type, and target language (of the code generation), respectively.

Each data constructor in Flow corresponds to a construction primitive for flows.

Kern. Turn a domain-specific program into a node – a kernel. Note that here a kernel itself is also considered a flow. In the description of PrimFlow it has a parameter with a type called k that represents a type the kernels stored in the network. A kernel is technically a wrapped program implemented in the assigned domain-specific language. This type must be implemented for each such language and passed to PrimFlow. Thus PrimFlow is an abstract type that requires information on how the kernel is represented. These requirements are characterized by the PrimKern type class that lists functions to extract details of the given kernels.

Kernels may be created by a lift function that has to be specified for each of the languages to be used in the dataflow network. A simple example for that may be given by wrapping a Feldspar function. Here, the lift function is called liftFeld that implements the previously introduced Liftable type class. The additional technical details will be revealed in Subsection 5.3.1.

kernel = liftFeld f
where
  f :: (Data Int, Data Int) -> Data Int -> Data Int
  f _ input = 2 * input

The wrapped f function masks all of the global parameters (the parameter with type (Data Int, Data Int), consisting of two integer values), and doubles the value it received as an input.

Seq. Sequence flows. That operation places flows in a sequential ordering. However, as we will see it later, this does not necessarily imply serialization of the execution. Sequentially connected flows may communicate by passing messages, therefore they may operate independently of each other in an asynchronous fashion.

It is formulated by the -- operator.
As the signature shows, it can only connect flows with the correct type: output type of flow $A$ must match with the input type of flow $B$ in the expression below, as well they have to use the same target language.

\[
\text{flow} = A \rightarrow B
\]

where $(A,B) = \ldots$

**Fork.** Split flow output. It offers a way to explicitly parallelize a computation by telling how to split the streamed data. It is implemented by the $\langle \langle$ operator.

\[
\langle \langle \rangle : (\text{Backend } t, \text{FlowType } a, \text{FlowType } b_1, \text{FlowType } b_2, \text{FlowType } c_1, \text{FlowType } c_2, \text{FlowType } d) \\
\rightarrow \text{Flow } a (c_1,c_2) d t \rightarrow (\text{Flow } c_1 d_1 d t, \text{Flow } c_2 b_2 d t) \\
\rightarrow \text{Flow } a (b_1,b_2) d t
\]

Tuples have a special meaning in the Flow language. Programs producing a tuple are considered *splitters* and programs taking a tuple are considered *mergers*.

\[
\text{flow} = A \langle \langle (B,C)
\]

where $(A,B,C) = \ldots$

Similarly to the sequencing operation, the corresponding types must match. In the example above, output types for $A$ ($\alpha_0, \alpha_1$) must match the input types of $B$ ($\alpha_0$) and $C$ ($\alpha_1$).

Note that due to the frugality of the basic building blocks, the opposite direction of forking may also be expressed without introducing any additional primitives. It only requires Fork and Seq: first we split the stream, then join the resulting substreams by concatenating a merge operation.

For convenience, the $\rangle\rangle$ operation could be created as an alias for such combinations.

\[
\text{flow} = A \langle \langle (B,C) \rangle\rangle \langle\langle D
\]

where $(A,B,C,D) = \ldots$

**Skip.** Bypass flow processing, no program is inserted into the node. It is merely a technical combinator that can be used to build up advanced connections.

It is represented by the split function whose type is similar to the type of a single kernel. This similarity is not a coincidence as Skip refers to an empty kernel. Though that could be even described by a simple identity program (that maps input to output without any modification), it may be worth promoting this special function to a primitive so that it can be optimized away during compilation.

\[
\text{skip} : (\text{Backend } t, \text{FlowType } a, \text{FlowType } c) \rightarrow \text{Flow } a a c t
\]
3.6.1 Examples

Let us take look at a few examples of how to describe graphs by using the presented combinators. Note that the programs in the nodes are not included here as our goal with the examples is to demonstrate how to represent various graph topologies with the defined operations.

Example 1. The following graph:

![Diagram of Example 1 graph]

can be described by this program:

```plaintext
example1 = a --> (b,c) --> d --> e
where (a,b,c,d,e) = ...
```

Example 2. The following graph:

![Diagram of Example 2 graph]

can be described by this program:

```plaintext
example2 = a --> (b,skip) --> c --> (d,e)
where (a,b,c,d,e) = ...
```

Let us note that dependency graphs may be represented in many other ways, Flow is just one example. As one may note by looking at the examples, Flow is basically a “point-free” or tacit language. In our opinion, this contributes to a succinct formulation of processing pipelines that may be enhanced further by adding more Arrow-like combinators [56], and support for loops. It is likely to be straightforward to use for programmers educated in functional programming, but others may also find them intuitive.
3.7 Abstract Programs

Given that the data dependency graph is described, there shall be also a way to compile it down to a less abstract platform in order to make it run on a given hardware. The intermediate representation used on that level will be called an abstract program for the rest of the document.

Abstract programs have two main components, tasks and channels. A task represents a run-to-completion operation to be performed on a single processing unit without interruption. This idea follows the typical hardware setup where processing units do not provide support or provide only limited support for preemptive execution (e.g. the Cell Broadband Engine)\(^1\). Such units usually excel in pure computational tasks with less branching or try to spare the expensive cost of context switching.

Tasks use channels to forward data between each other. They are like closed expressions or closures: the free variables (parameters) of the contained functions are bound by the values obtained from the channels and the actual state of the global configuration. Note that every task implicitly shares the same system-wide configuration (see Section 3.3). Channels are also the vehicles for establishing asynchronicity between nodes, although they also implicitly force serialization and deserialization of the communicated data at the same time. Hence the compiler may lose the opportunity to remove intermediate data structures between nodes and it has to implement certain optimization on passing data to avoid unnecessary copying. However, in some cases it is inevitable, and it may even result in better overall resource utilization than otherwise.

3.7.1 Graph Decomposition

Tasks and channels are derived from the graph by decomposition: each node paired up with its incoming and outgoing edges (as channels) is turned into a task. In Figure 3.3, such a decomposition is shown.

![Diagram of Graph Decomposition](image)

Figure 3.3: Decomposing a flow into tasks and channels.

In this example, the decomposition results in 6 tasks and 6 channels:

- **Task 1** has a source node that reads input data and puts to **Channel 1**.

\(^1\)It is most of the time intentional, i.e. it is by design. It is getting to be standard, with the emerge of space-shared parallelism.
– Task 2 has a sink node that reads Channel 6 and writes output data.

– Task 3 has a node that reads from Channel 1 and writes its result to Channel 2 and Channel 3.

– Task 4 and Task 5 has nodes that read from Channel 2 and Channel 3 and write to Channel 4 and Channel 5, respectively.

– Task 6 has a node that reads from Channel 4 and Channel 5 and writes its result to Channel 6.

As we mention above, source and sink nodes (marked with grey) are not part of the original graph as they are given by the user. Thus they automatically became “open”. For tasks, it implies that they do not have kernels associated. Hence they will be expressed in the target language we are compiling to. However, we will have to add some wrapping for those programs too, as there must be a way for them to access the channels. This will be described later, in Section 4.6.

Abstract programs can be characterized by the Program type, as a Cartesian product of set of tasks and channels for the given program. The sets are parameterized by the type of backend (t) and the global configuration (c).

type Program a b c t = ([Task c t], [ChanT])

3.7.2 Channels

In this representation, channels are described by their types. The ChanT is technically only a synonym for the previously introduced FlowT (meta)type, because our aim is to convert all data used by the kernels to this common representation. This facilitates making the solution independent of the actually used set of domain-specific languages.

type ChanT = FlowT

Channels themselves can be implemented in several ways that are highly dependent on the target platform. In the abstract program it is modeled as a regular Haskell list. Here channels are identified by their position in the list of channel types that is denoted by the CID type.

type CID = Int

Elements of the list are ordered by their channel number. For example, the type of the first channel is the zeroth element of the list.

channelType :: Program a b c t -> CID -> ChanT
channelType (_,channels) n = channels !! (n - 1)
3.7.3 Tasks

From the side of representation, tasks are nodes wrapped with information on the channels connected to them. That is specified by the Task algebraic data type. It has two type parameters: the first is the type of the global configuration, the second is the target language (to which the kernel is to be compiled), both of them are as we introduced them for the PrimFlow earlier.

```
type TID = Int

data Task c t
    = TaskI TID [CID]
    | TaskO TID [CID]
    | forall k l . PrimKern k l t => TaskK TID (PrimTask k l c t)
```

Each task has a unique identifier, represented by the TID type, which is derived from the identifier assigned to kernels in the Flow. That also determines an equality relation on tasks, given by an instance of the Eq type class.

```
instance Eq (Task c t) where
    t1 == t2 = taskId t1 == taskId t2
    where
        taskId (TaskI n _) = n
        taskId (TaskO n _) = n
        taskId (TaskK n _) = n
```

Further brief description of the data constructors is as follows.

**TaskI.** Stores a source node that is responsible for feeding the network with input data. It does not have a kernel assigned, because that must be provided in the target language and linked to the resulting binary (code generation) or expressed by a Haskell function (interpretation). Hence it contains information on the outgoing channels.

**TaskO.** Stores a sink node that consumes the output data generated by the network. It is similar to TaskI, in that it does not have kernel assigned, instead it must be provided in the target language. It contains information only on the incoming channels.

**TaskK.** Represents a generic task that can contain any kernel with input type \(a\) and output type \(b\) together with all the incoming and outgoing edges, identified by their channel number in the program. Note that kernels are existentially quantified as elements of the PrimTask type class, so their type parameters may be hidden to maintain a manageable formulation.

```
data PrimTask :: (* -> * -> * -> * -> * -> *) -> * -> * -> * -> * where
    K :: (PrimKern k l t, DSLType l a, DSLType l b, DSLType l c)
        => k a b c l t -> ([FlowT],[FlowT])
        -> [CID] -> [CID] -> PrimTask k l (T l c) t
```
The **PrimTask** has a kernel $k$, the information on its input and output types as *Flow* types, and the incoming and outgoing channels connected to the kernel. The order of the incoming channels must match the order of the input parameters of the kernel. The same applies to the outgoing channels and the kernel output parameters. That is, for any kernel $kern$ with type of source language $l$ and target $t$:

$$kern :: k \ a \ b \ c \ l \ t$$

with incoming channels $\text{incoming}$, outgoing channels $\text{outgoing}$, input types $\text{inpT}$ and output types $\text{oupT}$ in the wrapping **PrimTask** $t$:

$$t :: \text{PrimTask} \ k \ l \ c \ t$$

$$t = K \ kern \ (\text{inpT}, \text{outpT}) \ \text{incoming} \ \text{outgoing}$$

within the Program $p$:

$$p :: \text{Program} \ k \ a \ b \ c \ t$$

$$p = (t: \text{tasks}) \ \text{channels}$$

must satisfy the following:

and (map condition $[0..(n - 1)]$)

where

$$n = \text{length} \ \text{incoming}$$

$$\text{condition} \ i = \text{channels} !! (\text{incoming} !! i) == \text{inpT} !! i$$

and

and (map condition $[0..(n - 1)]$)

where

$$n = \text{length} \ \text{outgoing}$$

$$\text{condition} \ i = \text{channels} !! (\text{outgoing} !! i) == \text{outpT} !! i$$

### 3.7.4 Mapping Flows to Programs

We assume that every *Flow* can be mapped to a corresponding **Program**. The required transformations can be written as follows.

$$\text{toProgram} :: (\text{FlowType} \ a, \text{FlowType} \ b) \Rightarrow \text{Flow} \ a \ b \ c \ t \rightarrow \text{Program} \ a \ b \ c \ t$$

$$\text{toProgram} = \text{deriveProgram} \ . \ \text{toGraph} \ . \ \text{label}$$

this resolves to the following steps.

- **label**: Run the previously mentioned **State** monad and assign unique identifiers to the nodes in the graph.
toGraph: Use the generated node identifiers to build a regular directed graph from the dataflow network. To represent this graph, we use the standard Data.Graph Haskell data type [36]. It follows the mathematical definition of a graph, and it can be given as an ordered pair:

$$G = (V, E)$$

where $V$ is the set of vertices (nodes) and $E$ is the set of edges (lines). In this case, $V$ and $E$ denote the connected kernels and the connections between them, respectively.

```haskell
type Node = ID
type Channel = (Node, Node)
```

The network presented in Figure 3.3 can be given as a Data.Graph in the following expression. Note that the Data.Graph.graphFromEdges function lets us define a graph by its edges. Unit values () are inserted as nodes into the graph for technical reasons.

```haskell
g = graphFromEdges
  [ ((), 1, [3])
   , ((), 2, [])
   , ((), 3, [2,3])
   , ((), 4, [6])
   , ((), 5, [6])
   , ((), 6, [2])
  ]
```

Note that we must infer some of the connection details as the combinators of Flow do not give all information explicitly. For example, the >-- combinator is only a different name for the --> combinator thus it “disappears” when translated to a Flow.

It is done because it can be inferred in a deterministic way so there is no need to pass this information around. It helps to keep the number of basic combinators low and thus the implementation and specification is simple.

Consider the following flow:

```haskell
a --< (b,c) >-- d
```

it will be rewritten to this:

```haskell
a --< (b,c) --> d
```

Connections from a to b and c are explicit, whilst connections from b and c to d are implicit (see Figure 3.4).

It is because the Seq does not differentiate between regular subsequent flows and merging, it only checks for the matching types. That is, if d matches the type of the pair of the types of b and c, then it is allowed, and the proper connections will be derived.
Figure 3.4: An example of explicit and implicit (dashed lines) connections.

- **deriveProgram**: Identify the tasks and their connections as channels, put both of them into two separate lists.

  ```haskell
cannelsOf :: Graph -> [Channel]
cannelsOf = edges
```

Then we need to get the corresponding **ChanT**. We use the **PrimFlow** to learn what types the nodes in the network have. The definition is not detailed here.

```haskell
toChannel :: (FlowType a, FlowType b)
    => PrimFlow a b c t -> Channel -> ChanT
```

That is, the desired components are the results of the following.

```haskell
channelTypesOf :: (FlowType a, FlowType b)
    => PrimFlow a b c t -> Graph -> [ChanT]
channelTypesOf f g = map (toChannel f) (cannelsOf g)
```

- In order to derive tasks, channels must be numbered. It is implemented by assigning a unique natural number to each of them.

  ```haskell
tagChannels :: [Channel] -> [(Channel,CID)]
tagChannels = flip zip [0..]
```

We also need to learn how nodes in the graph are connected. It is implemented in the **connectionsOf** function. It takes the complete graph and an identifier for a node in that. Then it looks up the backward and forward connection (all incoming and outgoing edges), then resolves them to channel identifiers. Here the **Data.Maybe.mapMaybe** function is to used to map all the connections to a list of channels identifiers if found.
connectionsOf :: Graph -> [(Channel,CID)] -> Node -> ([CID],[CID])
connectionsOf g chans node =
  (mapMaybe resolve backwards, mapMaybe resolve forwards)
where
  backwards = ... -- all nodes that send to the given node
  forwards = ... -- all nodes that receive from the given node

resolve :: Node -> Maybe CID
resolve n = lookup (node,n) chans

Finally, we have to give the toTask function that finds the kernel assigned to a
given node identifier, and is used to prepare the list of tasks in the Program.

toTask :: PrimFlow a b c t -> Graph -> Node -> Maybe (Task c t)
toTask flow graph node =
  case (connectionsOf graph channels node) of
    ([],[]) -> Nothing
    ([],cids) -> Just (TaskI node cids)
    (cids,[]) -> Just (TaskO node cids)
    cids    -> findTask node flow cids
where
  channels = (tagChannels . channelsOf) graph

findTask :: Node -> PrimFlow a b c t -> ([CID],[CID])
         -> Maybe (Task c t)
findTask node f@(Kern id k) (icids,ocids)
  | node == id = Just (TaskK id (K k types icids ocids))
  where types = ... -- get types for kernel k

findTask node (Seq x y) c
  | isJust tx = tx
  | isJust ty = ty
  where (tx,ty) = (findTask node x c, findTask node y c)

findTask node (Fork x y z) c
  | isJust tx = tx
  | isJust ty = ty
  | isJust tz = tz
  where (tx,ty,tz) = (findTask node x c, findTask node y c, findTask node z c)
  = (findTask node x c, findTask node y c, findTask node z c)

findTask _ _ _ = Nothing

3.8 Semantics

Semantics for abstract programs can be given by simulating the operation of the repre-
sented flow. Simulation is defined in terms of an abstract machine M that has states and
transitions between the states. In the definition below, it is a Cartesian product of a state – the start state –, and a state transition function.

\[
\text{type } M \ a \ b \ c \ t = (\text{State } a \ b \ c \ t, \text{State } a \ b \ c \ t \rightarrow \text{State } a \ b \ c \ t)
\]

Note that \(M\) is defined over a target language (\(t\)) with input type of \(a\), output type of \(b\), and the global configuration type \(c\). This means that we can only get a concrete machine if those types are fixed.

The definition of \(\text{State}\) is as follows\(^2\).

\[
\begin{align*}
\text{type } \text{Execution} & = \text{Int} \\
\text{type } \text{Queue} & = [\text{Chunk}] \\
\text{type } \text{State } a \ b \ c \ t & = (([a],[b],[c]),([\text{Queue}],[\text{Task } c \ t],[\text{Execution}]))
\end{align*}
\]

Here we can see that \(\text{State}\) actually describes the contents of the abstract machine’s memory in a given execution step. It has the following components.

- \(([a],[b],[c])\): A 3-tuple of input, output, and configuration values – represented as Haskell lists – to be maintained while the machine is running. As we will see later, the input list contains all the (potentially infinite) values to be processed by the machine, while the output list is empty in the start state.

- \([\text{Queue}]\): State of the channels. It is represented as a list of message queues that are lists of values of the Flow type (\(\text{Chunk}\)). It works like a First-In-First-Out (FIFO) queue, and each of the queues can store arbitrary number of elements.

- \([\text{Task } c \ t]\): List of tasks to be executed. It is a constant list that comes directly from the abstract program to be run.

- \(\text{Execution}\): Index of the task (in the previous list above) to be executed in the next step. It may be set by a scheduling algorithm that represents a choice from the tasks available to run.

Operation of the abstract machine defined above can be characterized by a function that takes a flow to be run on a given input and configuration data, and returns a series of states. Note that input and configuration values, like states, are potentially infinite in number. Furthermore, it implies that the machine shall never stop. \(^3\)

\[
\text{simulate } :: (\text{FlowType } a, \text{FlowType } b, \text{FlowType } c) \\
\rightarrow \text{Flow } a \ b \ c \ t \rightarrow [a] \rightarrow [c] \rightarrow [\text{State } a \ b \ c \ t]
\]

simulate flow input config = iterate op s
where
  op = schedule . step
  s = inject ((prepare . toProgram) flow) input config

As we can see, the execution starts with a start state (\(s\)), and it repeats two main steps (\(op\)): finding the task to be run (\(\text{schedule}\)), running the chosen task (\(\text{step}\)).

\(^2\)Not to be confused with the \(\text{State}\) monad.

\(^3\)Which fits to our model as we are describing an ever-running system here.
- s: It is a regular state that is filled up with some initial input and configuration
data via the functions prepare and inject.

\[
\text{prepare :: Program } a \ b \ c \ t \rightarrow \text{State } a \ b \ c \ t
\]

\[
\text{prepare (program,queues) = ((I,I,,I),queues,tasks,0)}
\]

\[
\text{inject :: State } a \ b \ c \ t \rightarrow [a] \rightarrow [c] \rightarrow \text{State } a \ b \ c \ t
\]

\[
\text{inject ((_,output,_),internal) input config =}
\]

\[
((\text{input, output, config}), internal)
\]

Note that the flow is also mapped to an abstract program by toProgram. This
representation provides information on the tasks and channels.

- schedule: This is the function that changes the execution index after each step.
The easiest method for this is the round robin algorithm that simply increases the
index at each occasion, modulo the number of tasks.

\[
\text{schedule :: State } a \ b \ c \ t \rightarrow \text{State } a \ b \ c \ t
\]

\[
\text{schedule (w,(queues,tasks,execution)) = (w,(queues,tasks,execution'))}
\]

\[
\text{where}
\]

\[
\text{execution’ = (execution + 1) ‘mod’ numTasks}
\]

\[
\text{numTasks = length tasks}
\]

Note that there is only a single processor in this model. That is, we have only one
execution index and only one task is executed per step. This is acceptable, as we
will see later, that parallel execution of tasks only has affect on the performance but
not on the result itself. For the formal model here, performance is not considered.

- step: Performs a previously scheduled step, this is the transition function of the
abstract machine.

\[
\text{step :: (FlowType } a, \text{ FlowType } b, \text{ FlowType } c)\]

\[
\Rightarrow \text{State } a \ b \ c \ t \rightarrow \text{State } a \ b \ c \ t
\]

\[
\text{step (w,(queues,tasks,execution)) = (w',(queues',tasks,execution'))}
\]

\[
\text{where (w',queues') = st (tasks !! execution) (w,queues)}
\]

### 3.8.1 State Transformers

We have seen earlier that there is a helper function name \( \text{st} \). It lets us to specify how
the different task types (introduced earlier) should behave.

\[
\text{st :: (FlowType } a, \text{ FlowType } b) \Rightarrow \text{Task } c \ t \rightarrow (([a],[b],[c]),[Queue]) \rightarrow (([a],[b],[c]),[Queue])
\]

But before discussing the details, we will introduce a few other helper functions for our
convenience.

- popFront: It is a function that takes a message queue identifier, the queues in a
state, and pops the first element of the requested queue.
popFront :: QID -> [Queue] -> [Queue]
popFront _ [] = []
popFront 0 (q:qs) = tail q : qs
popFront n (q:qs) | n > 0 = q : popFront (n - 1) qs

pushBack :: QID -> Chunk -> [Queue] -> [Queue]
pushBack _ _ [] = []
pushBack 0 e (q:qs) = (q ++ [e]) : qs
pushBack n e (q:qs) | n > 0 = q : pushBack (n - 1) e qs

with and \(\triangleright\triangleright\): Minor technical functions to make the implementation more readable.

\[
\begin{align*}
\text{with } x y &= y x \\
x \triangleright\triangleright y &= y \cdot x
\end{align*}
\]

In the implementation of \(\text{st}\), we then identify the possible versions of tasks, and assign semantics to them one by one.

- A two-way task with a function with one input and one output channel. Take an element from the input queue (iq), get the current value of the global configuration, run the kernel with them, finally place the result (y) in the output queue (oq).

\[
\begin{align*}
\text{st } \text{(TaskK } _ \text{(K } k _ \text{ [iq] [oq]})\text{ all@((input,output,config),queues)} &= \\
\text{case (queues } ![\text{iq}, \text{config}) \text{ of} \\
(x:\_\_, \text{c:cs}) \rightarrow ((\text{input, output, cs),} \\
\text{with queues (} \\
\text{pushBack oq y } \triangleright\triangleright \\
\text{popFront iq} \\
\))} \\
\text{where } y &= (\text{toFlowType } \cdot \text{run } k \text{ . fromFlowType}) \text{ x} \\
_\_ \rightarrow \text{all}
\end{align*}
\]

- A two-way task with a function with one input and more output channels. Take an element from the input queue (iq), get the actual configuration value, run the kernel with them, and place the split results (y1 and y2, wrapped with CH\_N) in the output queues (oq1 and oq2).

\[
\begin{align*}
\text{st } \text{(TaskK } _ \text{(K } k _ \text{ [iq] [oq1,oq2]})\text{ all@((input,output,config),queues)} &= \\
\text{case (queues } ![\text{iq}, \text{config}) \text{ of} \\
(x:\_\_, \text{c:cs}) \rightarrow ((\text{input, output, cs),} \\
\text{with queues (} \\
\text{pushBack oq1 y1 } \triangleright\triangleright \\
\text{4Haskell has a function with the same symbol already: the sequencing operation in Monads. Well, we are to express something similar but simplified.}

42
- A two-way task with a function with more input and one output channels. Take elements from the input queues (iq1, iq2), get the current value of the global configuration, run the kernel with them, then place the result (y) in the output queue (oq).

\[
st \ (\text{TaskK} \ _ \ (K \ k \ _ \ [iq1,iq2] \ [oq])) \ \text{all@}((\text{input},\text{output},\text{config}),\text{queues}) =
\]
\[
\text{case} \ (\text{queues} !! \ iq1, \ \text{queues} !! \ iq2, \ \text{config}) \ of \\
\quad (x1:\_, \ x2:\_, \ c:cs) \ -> \ ((\text{input},\text{output},cs), \\
\quad \quad \text{with queues (} \\
\quad \quad \quad \quad \text{pushBack oq} \ x \ >> \\
\quad \quad \quad \quad \text{popFront iq1} \ >> \\
\quad \quad \quad \quad \text{popFront iq2} \\
\quad \quad \)) \\
\quad \quad \quad \quad \text{where} \ y = (\text{toFlowType} . \ \text{run} \ k \ c . \ \text{fromFlowType}) (\text{CH}_N \ [x1,x2])
\]
_ -> all

- A one-way task that directly reads the machine input. Read an element from the machine's input queue (iq).

\[
st \ (\text{TaskI} \ _ \ [iq]) \ \text{all@}((\text{input},\text{output},\text{config}),\text{queues}) =
\]
\[
\text{case} \ (\text{input}, \ \text{config}) \ of \\
\quad (x:xs, \ c:cs) \ -> \ ((xs,\text{output},cs), \\
\quad \quad \text{with queues (} \\
\quad \quad \quad \quad \text{pushBack iq} \ (\text{toFlowType} \ x) \\
\quad \quad \)) \\
\quad \quad \quad \quad \text{all}
\]

- A one-way task that directly writes the machine output. Take an element from the output queue (oq), and write it.

\[
st \ (\text{TaskO} \ _ \ [oq]) \ \text{all@}((\text{input},\text{output},\text{config}),\text{queues}) =
\]
\[
\text{case} \ (\text{queues} !! \ oq, \ \text{config}) \ of \\
\quad (x:\_, \ c:cs) \ -> \ ((\text{input}, \ \text{output'}, \ cs), \\
\quad \quad \text{with queues (} \\
\quad \quad \quad \quad \text{popFront oq} \\
\quad \quad \)) \\
\quad \quad \quad \quad \text{where} \ \text{output'} = \ \text{output} ++ \ (\text{fromFlowType} \ x)
\]
_ -> all

Remarks:
For every case, if there is no information ready in any of the queues then the machine
does not produce output. That is, it does nothing other than moving between tasks
according to the actual scheduling algorithm.

Because the simulate function generates an infinite sequence of states, output can
be observed only by picking one of the states and extracting the actual state of the
machine output from it.

That is, the following expression

\[
eject ((\text{simulate flow input config}) !! n)
\]

gives the accumulated output of the machine in its \(n^{th}\) step. Here, eject is a simple
function that extract the output from the state.

\[
eject :: \text{State a b c t} \rightarrow [b]
eject (\text{(_,output,_,)_,}) = \text{output}
\]

When there are more open nodes in the network, the machine input and output must
be adjusted according to them. Multiple inputs and outputs may be represented as
n-tuples, where only one of them is ever read or written depending on which open
node was being processed.

### 3.9 Summary

In this chapter, we have introduced the Flow language as tool for building models of
operating systems. The Flow is technically a glue language that is embedded into Haskell,
and it supports composition of small programs written in other embedded languages
into larger ones. The little languages to be used to express those small programs are
considered domain-specific, which means they usually implement some model for a set of
operations that specific to a given application or problem domain. This approach helps
to describe certain algorithms in a language that is better suited to the problem to be
solved and eventually supports efficient code generation. Furthermore, given that there
is a pure semantics implemented for the language, it can be seamlessly extended with the
constructs of the Flow language to build or model larger systems, e.g. operating systems
that way.

As we have learnt here, each such small program has to be “lifted” (wrapped) to a
primitive Flow node and then it may be further combined with other similar nodes with
the presented combinators. The result of the combination is a directed graph that records
data dependency relations between the programs, hereby building a dataflow network.
Those networks were considered special, because they are repeatedly run with incoming
data through their source nodes to produce data through their sink nodes. This may be
interpreted as a simple model of a main loop which is to receive and answer requests from
an environment.

Lifting is restricted: not all the little languages are supported, only the ones that
are able to implement the operations in the PrimKern type class. At the moment, not
all the formulated requirements are exploited, but we will continue with that in the
next chapter. The domain-specific types associated with each language also have to be
mapped to a set of common types, called FlowTypes, which is important, because the Flow is independent of the languages used in the created network, therefore it must also provide an independent means of communication between the nodes. Because of that Flow is not aware of the communication protocols the nodes use to transfer data.

Finally, we have given the semantics for a Flow program through its abstract counterpart, an abstract program. That abstract program can be run with a corresponding abstract machine that works with tasks and channels. Tasks are tightly connected to channels as they receive their input from them and they send their output to them. Besides the input, tasks may access and use the elements of a global set of parameters for their work. Although such information may be part of the input, it makes sense to handle them specially. For example, so that tasks may share parameters.
Chapter 4

Real World Execution of Flows

Besides the naïve abstract interpretation of Programs, data dependency graphs may be prepared for execution on multiple processing units, i.e. on top of real hardware. A trivial solution would be to launch an execution thread for each of the tasks (then no further processing would be needed). Although it is not considered a good practice to spawn an arbitrary number of threads in general. Doing so may unnecessarily waste resources. And, if context switching is even possible, it may lead to more time being spent context switching. For better performance it is instead recommended that one not to keep more threads running at a time than the number of the processing units in the hardware.

Restrictions such as the fixed number of threads or running tasks to completion contribute to a characterization of the run-time system support. Thus there is a mechanism by which to escape from the concept of preemptive multitasking and replace it with concurrent cooperative multitasking. It is worthwhile avoiding preemption as it has a number of disadvantages. Context switching required for preemption takes time, since run time of task is split into small scheduling intervals (quantums), interrupted by a call to the scheduler to decided what to continue with next. And if indeed a switch occurs between tasks, possibly a new set of data has to be placed into the cache, extending the time required for completing the operation. This also displaces from the cache the working set of the preempted task. Asynchronous interruption of tasks also requires the programmer to protect all shared resources with locks which in turn increases the costs due to the implied synchronization.

In addition, the scheduling solutions usually employed in operating and run-time systems are too general for our purposes. Since the programs to be run are split into runnable sections dynamically and without any knowledge of the current program, they can only rely on generic heuristics, e.g. interactivity. When the units of scheduling are adapted better to the domain of the application, the opportunity arises to make better scheduling decisions based on how the programs to be run are split or what their actual states are.

4.1 Synopsis

The focus of the chapter is to investigate how to construct an execution scheme for tasks whose performance may be sufficiently efficient in a real-world setting. First we will discuss how to implement execution of each individual task and how to organize input and output data for tasks. That tasks can be put into pools that is one of the main concepts of the chapter. It will then be followed by a detailed description of how a task pool might
be handled in the presence of multiple execution units, as well as example of the problems it can cause. Memory management will also be taken under consideration. As we will show as the study progresses, the discussion of possible task execution scenarios leads to introducing explicit hinting on how to schedule tasks, together with the definition of task pools. The results will help us to define a parallel version of the previously introduced abstract programs and the associated machines. We will thereby extend the semantics and the simulation of Flow programs for parallel execution.

For the remainder of the chapter, the code generation for Flow programs will be discussed. By explaining the details of the proposed code generation procedure, the previously omitted components of the PrimKern type class will be presented through the introduction of a three-stage compilation process, together with a simple algorithm to reduce the potential for code bloat. Note that in relation to the PrimKern type class, another previously hidden class, Backend will be explained. The Backend class has already appeared in the definition of the Flow combinators, as well as a constraint on the languages to be used for the nodes. As it will turn out it prescribes a crucial property: the Flow can be only compiled to the target language if there is a compiler for all the languages to that language. Besides that, compilation of both the regular and nodes without little programs will be explained. In connection with the nodes without a program we will also touch on the question of how to deploy a Flow system.

4.2 Executing Tasks

As we have previously indicated, we will introduce the traditional thread pooling pattern, in which a fixed number of threads are started to perform a number of tasks which are typically organized as a queue. Threads are considered “worker processes” or simply workers. Workers grab tasks from a task queue – or a pool –, execute them, then return the results to somewhere.

Since workers may pull in new tasks only when they are done with the previous one, this scheme provides for sharing the load between the processing units dynamically. And, due to its nature, it does not have to be specified in the program itself. Instead it can be incorporated as an implicit part of the model. This approach also plays well with heterogenous computer systems where there is a general-purpose control unit to distribute and dispatch the work to units that are heavily optimized for data processing operations.

Execution of a task involves reading from the input channels, running the encapsulated program, and finally writing the result to the output channels. As we have already seen in the case of abstract programs, channels have to work as FIFO queues. Although in practice they may be of fixed length. In this setting, channels block writing when they are full, and they block reading when they are empty. When any of the task’s (input or output) channels is blocked the contained kernel itself cannot be run so its execution is deferred.

4.3 The Pool of Tasks

As we have mentioned above, in order to schedule tasks for execution, they are organized into a task pool. A task pool is a set of tasks that may be run independently of each other, possibly in parallel. Workers are assigned to the pool and they get their jobs to be
handled from there. Because there may be more workers for the pool, workers may share work between each other dynamically. Since tasks are closed over their parameters, any worker may run any task without changing the output.

A task pool may be given formally as a list that contains tasks to be executed. It is parametrized over the global configuration (c) and the backend, i.e. the concrete representation (t).

type TaskPool c t = [Task c t]

There are two primary operations for the task pool: take and drop. We are going to use the STM (Software Transactional Memory) [29] and the IO monads here to define their abstract semantics in Haskell. The STM monad implements composable atomic memory transactions – all operations in an atomically block will be completed at once or retried when the shared resource becomes available again. Thus, the atomically functions assumes proper synchronization for accessing the shared variables. Shared data is described as TVar s (transactional variables) that can be read by readTVar and written by writeTVar.

There is also a simpler version of the previously introduced State type is given below to simplify the description of the functions to be presented.

type SimpleState a b c t = ([a],[b],[c],[Queue])

The take operation gets a task from the task pool. If there is no task found in the pool then it returns Nothing. Note that it is written as a single STM block which may be composed further.

Formally:

\[
\text{take} :: \text{TVar} \left( \text{TaskPool} \; \text{c} \; \text{t} \right) \rightarrow \text{TVar} \left( \text{SimpleState a b c t} \right) \rightarrow \text{STM} \left( \text{Maybe} \left( \text{Task c t} \right) \right)
\]

take pool _ = do
t <- readTVar pool
  case t of
    (t:ts) -> do
      writeTVar pool ts
      return (Just t)
    _ -> return Nothing

The drop operation puts a completed task back to the pool. It shall always succeed as there must be enough place for tasks in the pool.

Formally:

\[
\text{drop} :: \text{TVar} \left( \text{TaskPool} \; \text{c} \; \text{t} \right) \rightarrow \text{TVar} \left( \text{SimpleState a b c t} \right) \rightarrow \text{STM} \left( \text{Task c t} \rightarrow \right)
\]

drop pool _ t = do
ts <- readTVar pool
writeTVar pool (ts ++ [t])

Based on the definitions above, semantics for the workers can be given as follows.

\footnote{The take and drop are standard list functions in Haskell, but we define our functions with the same name here.}
workerStep :: (FlowType a, FlowType b) => TVar (TaskPool c t) -> TVar (SimpleState a b c t) -> STM ()
workerStep pool state = do
  t <- take pool state
  s <- readTVar state
  case t of
    Just task -> do
      updateStateWith state (st task s)
      drop pool state task
    Nothing -> return ()

Every worker receives reference to a task pool and every worker accesses the same global variables, now they are summed up as a composition of tuples. First the worker tries to pick a task for itself: if succeeds then it executes it. Otherwise, the execution of the given worker is suspended and re-‐tried later to avoid busy waiting. Note that the suspension is covered by the semantics of the STM monad hence it is not mentioned here explicitly. The updateStateWith function is not detailed here – its purpose is to merge the values back to the global state that were changed locally.

Here the workerStep function describes a single step in the endless loop that the worker must implement.

worker :: (FlowType a, FlowType b) => TVar (TaskPool c t) -> TVar (SimpleState a b c t) -> IO ()
worker pool state = (forever . atomically) (workerStep pool state)

Note that any task may be chosen from the pool. It would be the job of a scheduler to pick a task for a worker, however, it is not required as it can be expressed by take and drop. We may say that let take select the first task (head) from the pool, while drop appends the completed task to the end of the pool. This way we get the regular round-robin scheme, see Figure 4.1.

```
Figure 4.1: Operation of a single worker over a task pool.
```

In case of more workers, the consequences of that behavior is not this straightforward, as tasks may be reordered in the pool based on their time required for execution. This may cause problems in scheduling because dependent tasks (that are connected to the given task via the channels) are executed in the wrong order. This may imply too many
Figure 4.2: Operation of multiple workers over the same task pool. Note that order of elements may change during parallel execution.

blocked tasks selected for execution, thus resulting in a massive slowdown for multiple workers.
This can be relaxed by building up more task pools by taking the dependencies between tasks into consideration. It can then be implemented by partitioning the tasks into layers. Each layer contains tasks whose parallel execution could have bad effects on each other. Therefore any of them may be selected for running in an arbitrary order, the chances of blocking will be minimal.

As an example, consider the four tasks in Figure 4.1. Let us assume that they are connected in the shape of a diamond, that is \( t_2 \) and \( t_3 \) depend on \( t_1 \), while \( t_4 \) depends on \( t_2 \) and \( t_3 \). In this case, it would be better to create two pools: one for \( t_2 \) and \( t_3 \), and one for \( t_1 \) and \( t_4 \) as shown in Figure 4.3. That way workers will not violate the dependency ordering for the tasks, and task pools do not have to be prepared for concurrent execution (e.g. no mutexes are needed) as there is only a single worker per pool.

Figure 4.3: Operation of multiple workers over multiple task pools.
4.4 Memory Management

In addition to processor time, tasks usually require a chunk of memory in order to operate. A generic layout for tasks is shown in Figure 4.4. There we can see that tasks work with input and output buffers. The former is used for reading values from channels (there may be more), and the latter is used for storing results of the wrapped program. Size of the buffers may be derived from the size of the type of values they store. It may be then multiplied by a chunk size if multiple values travel on the channel in a single pass – thus some space has to be reserved for them in the memory.

![Task layout](image)

Figure 4.4: Task memory layout in general.

The contained kernel itself may also have its own memory requirements. It is called heap in Figure 4.4, which acts as a temporary storage area when the program is run. Heap is passed to the program so it does not have to use its own stack for calculations that require large amount of memory. However, an exact size for this cannot be told in advance in the dataflow layer. Hence the kernel must provide hints via heapInfo function in the PrimKern type class (as mentioned earlier, see Section 3.5).

Tasks get memory assigned when they are executed by a worker, and workers will allocate memory only on their start. That is, workers must have enough memory allocated to be able to run any task from the pool they are working with as reallocating memory for each task may degrade their performance, and would also require some support for dynamic memory management. Assumed that sizes for the buffers and the program can be statically determined at compile time, it can be given how much memory is needed for a given worker. It can be expressed by $M(\cdot)$ as follows.

$$M(w) = \max \left\{ \sum_{i=0}^{n_k} S(I(k, i)) + \sum_{j=0}^{n_k} S(O(k, o)) + S(H(k)) \right\} \quad k \in K(w)$$

where $w$ is the worker, $k$ is a kernel from a set of kernels that the worker may run ($K(w)$), $S(\cdot)$ specifies the size for a given entity, $H(\cdot)$ gives all kernel allocations (size of the kernel heap), $I(\cdot, \cdot)$ and $O(\cdot, \cdot)$ gives the $n^{th}$ input and output buffers for $k$, respectively.

Note that because of the maximum function, certain kernels with high memory needs may give rise to suboptimal memory usage. For example, if there is only a few large-heap kernels then workers may allocate too much memory because they may want to run them.
That is, when memory requirements are not uniformly distributed among kernels, worker allocations will take the largest one that may waste resources. A solution for that would be to take this into consideration when tasks are partitioned into task pools – besides respecting their dependencies.

4.5 Scheduling

To address the problems that have been sketched up to, we propose three ways to enhance the model with programmable scheduling, both of which are expressed as extensions with some restricted language constructs.

4.5.1 Marking Boundaries for Kernels

Note that because the kernel is the unit of execution (Section 4.2), decomposing the Flow into kernels is crucial to the performance of the given program. Viz. that way the programmer gains some control over the granularity of the scheduling quanta as tasks are not interrupted. The programmer may choose to describe the application differently where the kernels are “fused” together, that is, all the intermediate data structures are removed as a result of the compilation. Thus there is no communication needed between the different processing units, the serialization and deserialization of data may be simply optimized away. Moreover, as the Flow program enables the programmer to work on a higher level, that is, the result is usually compact and the details are left to a code generator, it is relatively easy to reorganize the kernels.

With the possibility of describing a task pool in a similar high-level way, we get the concept that we may call “declarative scheduling”: the user specifies the manner in which the program is scheduled in terms of some high-level primitives without dealing with the low-level ones. Instead, those are added automatically as an increment of the combined use of the primitives.

4.5.2 Selectors

The basic idea of declarative scheduling is to mount a “selector” on the take operation. Here, it is expressed as a function over the elements in the task pool. It implies the following modifications to the earlier definition of take.

```haskell
take' :: TVar (TaskPool c t) -> TVar (SimpleState a b c t) -> Selector -> STM (Maybe (Task c t))
take' pool state f = do
  p <- readTVar pool
  case p of
    [] -> return Nothing
    xs -> do
      let (t,ts) = selectBy f xs state
      writeTVar pool ts
      return (Just t)
```

That is, the selector has the following type.
`data Selector :: * where
   Next :: Selector
   By :: Property -> Selector`

Note that it has been applied together with a `selectBy` function which is used to specify the abstract semantics for each of the primitives.

`selectBy :: Selector -> [Task c t] -> SimpleState a b c t
   -> (Task c t, [Task c t])`

As is shown above, the `Selector` type contains only an abstract description of how to pick the next task for execution. A brief explanation of the constructors is as follows. Please note however, that there additional similar functions may be introduced later.

**Next.** Pick the next available task in the pool independently of the current state. This is the original behavior of `take` where the head of the task pool list was chosen.

`selectBy Next pool _ = (head pool, tail pool)`

**By.** Select a task from the pool based on a certain property, described by the `Property` type.

`selectBy (By p) pool state =
  (snd $ head mins, map snd $ tail mins ++ misc)
where
  (mins,misc) = partition (\(s,t) -> s == m) scoredPool
  m = minimum (map fst scoredPool)
  score t = (byProperty p state t, t)
  scoredPool = map score pool`

For the definition of `selectBy`, we used the `partition` function from `Data.List` standard Haskell module.

Note that we could have used only the `By` combinator as a primitive operator for expressing selector functions. The distinction made is rather categorical: the current position of a task in the pool (which the selection would have based on) is not a property in the same sense as e.g. the length of its input queue. Therefore expressing it as, for example a property would also make the recognition of the given operator harder at code generation hence that cannot be simplified to get an optimized translated version.

In the above description, the `byProperty` function was used to define semantics for the evaluating a certain property for tasks, and it has the following signature.

`byProperty :: Property -> Task c t -> SimpleState a b c t -> Int`

It is based on the `Property` type that is to list all the properties that may be calculated for the tasks. An example of such a property is the maximal input queue length.

`qLen :: Property`
which is represented by a data constructor in the Property type.\textsuperscript{2}

QLen :: Property

whose semantics is given as follows.

\[
\text{byProperty QLen task } (_{-}, _{-}, _{-}, \text{queues}) = \\
\quad \begin{cases} 
\text{TaskI } _{-} & \rightarrow 1 \\
\text{TaskO } _{-} qs & \rightarrow \text{minimum } (\text{map } (\text{length} . (\text{queues }'!!)) \text{ qs}) \\
\text{TaskK } _{-} (K _{-} _{-} qs _{-}) & \rightarrow \text{minimum } (\text{map } (\text{length} . (\text{queues }'!!)) \text{ qs})
\end{cases}
\]

Here we use the received SimpleState value as the queueing information associated with each task that has to be retrieved from there with indexing the global queue table (queues).

Source nodes (TaskI) have to be handled specially: they do not have input queues and it is assumed that they can always provide data. Therefore a constant score is assigned to them. The selected constant here is 1 that means the flow sips input only if there is no data inside to be forwarded, giving priority to data processing operations that way.

Because we want to get the task with the maximum queue length, the calculated score has to be inverted. To implement that, simple property transformers can be introduced for the user’s convenience.

\[
\text{minimumOf} :: \text{Property } \rightarrow \text{Property} \\
\text{minimumOf } p = p
\]

\[
\text{maximumOf} :: \text{Property } \rightarrow \text{Property} \\
\text{maximumOf } p = \text{Invert } p
\]

It requires the addition of the Invert constructor to the Property type, with the following semantics.

Invert :: Property \rightarrow Property \\
byProperty (Invert p) task = negate (byProperty (p task))

So choosing a task by the maximal queue length can be written as follows.

\[
\text{maxQLen} = \text{maximumOf } \text{qLen}
\]

4.5.3 Creating Task Pools

Besides selection, it is also possible to control how task pools are organized. The purpose is to cover the problems of respecting data dependency between tasks to minimize the chances of picking an already blocked task (as observed in Section 4.3) and uniform distribution of tasks by memory requirements (see Section 4.4).

Controlling organization of task pools requires the tasks to be identified somehow inside the application. Since the user has no exact knowledge on how tasks are derived, there must be a tool for tagging flows provided instead.

\textsuperscript{2}The Property may contain other constructors to characterize other supported properties.
type Pool = (ID, Selector)

As we will see through the section, task pools are described as a Cartesian product of an identifier (ID) and a selector (Selector). The former tells which task pool we are talking about at the given point of the flow, and the latter assigns a scheduler – as a function of the selector – to the pool.

In order to tag, pools have to be created first by the `createPool` function. There the user has the freedom to pick an identifier for the pool to be created. This will be exploited later.

```haskell
createPool :: ID -> Pool
createPool id = (id, defaultSelector)
```

Here a default selector is assigned to each fresh pool which is actually the previously characterized default behavior of the `take` function.

```haskell
defaultSelector :: Selector
defaultSelector = Next
```

The `scheduleBy` function is used to override the semantics of the selector for one of the pools.

```haskell
scheduleBy :: Selector -> Pool -> Pool
scheduleBy s (id,_) = (id,s)
```

In the high-level representation (like the one we have presented in Section 3.2), we can only refer to nodes so there we introduce an operator to implicitly annotate tasks to be created through their parent sub-flow. In our opinion, it is safe to introduce, as tasks are deterministically derived from the nodes in the flows (see Subsection 3.7.4).

```haskell
infix 2 #=

(#) :: (Backend t, FlowType a, FlowType b, FlowType c) => Pool -> Flow a b c t -> Flow a b c t

The #= operator annotates the given sub-flow of the Flow graph, assigning every derived task to the specified pool (together with all of its properties) during the composition. Such annotations may then be nested in a fashion similar to nesting mathematical sets.

```haskell
pool1 #= flow1 --> (pool2 #= flow2 --> flow3) --> flow4
```

In the example above, tasks created from `flow1` and `flow4` are added to `pool1`, while tasks from `flow2` and `flow3` are added to `pool2`.

Tagging in the Flow language is implemented by appending the following constructor function to the core constructs (see Section 3.6).

```haskell
Pooled :: (Backend t, FlowType a, FlowType b, FlowType c) => Pool -> PrimFlow a b c t -> PrimFlow a b c t
```
With Pooled, the pool that contains the given task can be determined by finding the first pool tag going upwards from the associated kernel in the flow.

Note that they are optional to use. If none of them is applied, the compiler shall select one of the built-in heuristics for organizing the tasks into pools and controlling the take and drop operations. It is because the toProgram function (see Subsection 3.7.4) can be extended with a single step in the following way.

\[
\text{toProgram} = \text{deriveProgram} . \text{toGraph} . \text{label} . (\text{createPool} 0 #=)
\]

With that, we assign the pool 0 to the entire flow to be compiled, therefore we are creating a “root pool” that contains all the tasks. And due to the way of finding pools for tasks it technically means that if there is no tag added for the given sub-flow then its associated pool will be root pool by default. That also applies to implicit source and sink nodes as well. They will be always in pool 0.

Hence it may happen that we want to re-use an existing pool. We add the usePool function to express a partial pool label, where only the identifier is given.

\[
\text{usePool} :: \text{ID -> Pool}
\]

\[
\text{usePool} = \text{createPool}
\]

As the definition shows, it is actually a different name for createPool.

Note that both the partitioning of tasks and the specification of selectors may be derived automatically. All that has to be done is to insert the pool tags by a given algorithm instead of doing it manually.

### 4.5.4 Example

To briefly summarize the concepts introduced above, a simple example is given here. Consider the following dataflow network:

It can be described in the following way:

---

3The most trivial heuristics is to put all tasks in the same pool.

---
pool0 = scheduleBy maxQLen (usePool 0)
pool1 = createPool 1

f = pool1 #= f1 --< (pool0 #= f2, pool0 #= f3) >-- f4

That is, we partition the flow into 2 task pools:

- **Pool 0** contains the implicit source and sink nodes together with both branches of the parallel composition \((f2, f3)\). Here the scheduling policy is overridden by picking tasks based on their maximal queue length.
- **Pool 1** contains the splitter \((f1)\) and the merger \((f4)\) programs.

### 4.5.5 Parallelizing Abstract Programs

Through the definition of how to create task pools, it is also possible to describe a parallel abstract machine for execution. In order to do that, the basic definitions from Section 3.7 and Section 4.3 must be reworked to some extent.

The **TaskPool** type – that we have used before to describe a set of tasks – is now restructured to store the corresponding pool tag (Pool) and the set of tasks ([Task c t]) there. There is also the **Execution** type added to denote which is the next task to be executed in the given task pool.

```haskell
type TaskPool c t = (Pool, [Task c t], Execution)
```

The **Program** type is modified to store a set of task pools instead of a single one that have been used previously.

```haskell
type Program a b c t = ([TaskPool c t], [ChanT])
```

The **Flow** programs tagged with pool information (see the **Pooled** constructor) can be mapped to **Programs** corresponding to the definition above if the **toTask** and **findTask** functions are modified as follows:

```haskell
toTask :: PrimFlow a b c t -> Graph -> Node -> Maybe (Pool, Task c t)
toTask (Pooled pool flow) graph node =
  case (connectionsOf graph channels node) of
    ([], []) -> Nothing
    ([], cids) -> Just (pool, TaskI node cids)
    (cids, []) -> Just (pool, TaskO node cids)
    cids -> findTask pool node flow cids
    where
      channels = (tagChannels . channelsOf) graph

toTask _ _ _ = Nothing
```

```haskell
findTask :: Pool -> Node -> PrimFlow a b c t -> ([CID], [CID])
         -> Maybe (Pool, Task c t)
findTask pool node flow (icids, ocids)
  | node == id = Just (pool, TaskK id (K k types icids ocids))
```
where types = ... - get types for kernel k

findTask pool node (Seq x y) c
| isJust tx = tx
| isJust ty = ty
where (tx,ty) = (findTask pool node x c, findTask pool node y c)

findTask pool node (Fork x y z) c
| isJust tx = tx
| isJust ty = ty
| isJust tz = tz
where (tx,ty,tz) = (findTask pool node x c, findTask pool node y c, findTask pool node z c)

findTask _ node (Pooled pool flow) c = findTask pool node flow c

findTask _ _ _ _ = Nothing

and post-processed by the following function:

map (\l -> ((fst . head) l, map snd l, 0)) .
groupBy ((==) 'on' poolId) .
sortBy (compare 'on' poolId) tasks
where
    poolId = fst . fst
tasks = ... -- all the tasks in the decomposed graph

It sorts, then groups the tasks by their associated pool identifier (determined by toTask), and finally maps the groups to task pools.

In the next step, the State type has to also follow the changes above.

type State a b c t = (([a],[b],[c]),([Queue],[TaskPool c t]))

With State modified, the adapted semantics is then given by a renewed definition of schedule and step.4

schedule s@(w,(queues,taskpools)) = (w,(queues,taskpools'))
where taskpools' = map (doSelect s) taskpools

In the definition above, the doSelect function is responsible for calling selectBy which picks a task guided by the current task pool’s selector (f). The elemIndex Haskell function helps to determine which index the selected task has within the pool.

doSelect :: State a b c t -> TaskPool c t -> TaskPool c t
doSelect ((wi,wo,wc),(qs,_)) (p@(_,f),tasks,i) = (p,tasks,i')
where
    Just i' = elemIndex t tasks
    (t,_) = selectBy f tasks (wi,wo,wc,qs)

4Assumed that utility functions, e.g. prepare are also modified accordingly. They are not mentioned here as they are not important to the reasoning.
Note that doSelect is mapped over all task pools, meaning that schedule sets the execution index for all pools at the same time.

Similarly, the step is changed in the same manner.

\[
\text{step } (w,(qs,taskpools)) = (w',(qs',taskpools))
\]

where

\[
(w',qs') = \text{foldr } f (w,q) \text{ taskpools}
\]

\[
f (_,tasks,i) = \text{step'} (\text{tasks} !! i)
\]

Here the new state of the world and queues is a result of folding over the per-pool selected tasks, which corresponds to a simultaneous execution.

### 4.6 Code Generation

Code generation depends on the target language and platform. Our approach does not restrict the choice of the target language, the only requirement is that there must be a mapping to it from the abstract program representation. For the rest of the document, we will use the C programming language as an example.

To start the discussion on the code generation, let us continue with detailing the functions in the PrimKern type class presented in Section 3.5.

```haskell
class Backend t => PrimKern k l t where
  run :: (DSLType l a, DSLType l b, DSLType l c) => k a b c l t -> T l c -> T l a -> T l b
  compile :: k a b c l t -> ID -> (Types t, Name, Body t)
  finalize :: k a b c l t -> Name -> Body t -> Definition t
  heapInfo :: k a b c l t -> Types t
```

The compilation of programs is divided into three phases due to the redundancy elimination (Subsection 4.6.1) and the generation of task wrapper code (Subsection 4.6.2). First, the compile function is responsible for turning kernels to target-language programs with the given identifier, as determined by the Backend instance of the target. Although it is even possible to combine programs expressed in multiple domain-specific languages, the target language must be the same for all of them.

```haskell
class Backend t where
  type Types t
  type Body t
  type Definition t
```

The Backend type class groups the types used during the code generation – which may be taken as an abstract description of a simple subroutine-oriented programming language. It consists of optional type declarations (Types t) that may be required for the routine body (Body t) then finally turned into a definition (Definition t) in the target language.

For example, a C backend may be defined by the following instance.
instance Backend C where
  type Types C = CTypes
  type Body C = String
  type Definition C = String

As is shown above, the function body and definition are technically represented as strings in the C backend. The unusual type here is CTypes (which is not detailed here further) that may be reasoned as follows. It is used to collect all the type definitions from the resulted function as multiple definitions of the same type has to be eliminated somehow – otherwise it may give us incorrect C code that cannot be compiled. As a consequence, it implies checking for equivalence on type definitions. Rendering those definitions as strings then parsing them back may result in a suboptimal solution, so it is more logical to ask for their abstract representation, and that is what CTypes captures here.

4.6.1 Eliminating Redundancy

In the second phase, the finalize function is invoked to put the previously translated body and its name together to form a complete definition. It is because of the redundancy elimination: During the code generation phase, it is not unusual for the same kernel to be used in multiple places within the same flow. That is, tasks may share the kernels they contain. This is a consequence of the sharing properties of Haskell expressions. However, if this phenomenon goes unnoticed, the generated code may easily suffer from excessive code growth. Hence it is advised to add some support for elimination of redundant code. The proposed algorithm is as follows.

- Create a look-up table for the compiled code. Let this table store all the previously compiled program bodies as plain strings. Initially it is empty.
- When code generation for a kernel is finished then look it up in the table.\footnote{The efficiency of the search may be improved by using a specialized data structure, e.g. a balanced search tree.}
  - If it is found then return the position in the table and generate only a call to the given kernel. Kernel names can be composed in a systematic way, using a template, e.g. \texttt{kern\_N} where \texttt{N} is the position in the table.
  - Otherwise insert the generated body into the table, add the body to the result, and place a reference pointing to it within the code generated for the containing task.

When generating code for the kernel body, the name itself should be omitted as it may not be specific to the kernel in question. Name is added only after the redundancy elimination algorithm has been run. Note that there could be hashes computed for kernel bodies.\footnote{However, we should be careful with hashes as they may collide.} The presented method is better than if the user had to manually name kernels in the graph. It is less prone to errors (as it is automatic) and it may spot duplicates even if they resulted from two different high-level programs. Thus programs are first compiled
without their names. Then the bodies can be checked for being redundant, and if they are not, then they are combined with their names. Note that names may be derived from unique identifiers of nodes.

4.6.2 Wrapping Nodes

The generated target-language code for tasks must implement the semantics presented in Section 3.8. Tasks must also be suitable for execution by the workers. As we have described earlier in Section 4.2, workers “grab” tasks from a pool, and try to execute them. During the execution, the generated wrapper for tasks takes care of the management of the input and output channels (as it has been shown in the semantics), so the only information has to be communicated to the kernel to be run is the place of the chunk reserved for running the tasks in the memory (see Section 4.4) and the actual state of the global configuration.

![Diagram of a worker with tasks replaceable in run time.](image)

Figure 4.5: Graphical view of a worker with tasks replaceable in run time.

Hence, in the third phase, some outer glue code for the kernels is generated to adapt them to the run-time environment of the created dataflow network. The goal is build up a counterpart of the wrapping shown with DSLType in the target language. That is, kernels still have to maintain a mapping between their source language and the Flow types in the backend. Thus only a wrapper function similar to the one below has to be written for each domain-specific language.

```haskell
wrapper :: Backend t =>
        Identifier -> (CfgType,[FlowT],[FlowT]) ->
        Identifier -> ([FlowT],[FlowT],[FlowT]) ->
        Definition t
```

The first set of parameters represent type information for the outer interface, i.e. types for the global configuration (CfgType), and the input and output edges in the graph (FlowT), with a name for the corresponding C function (Identifier). The second set of parameters represent type information for the inner interface, i.e. types that the kernel to be wrapped has: input, output, and heap (FlowT), and its name in C (Identifier). The
latter is where the last function of PrimKern, heapInfo is used. The result of heapInfo is to tell the compiler what type of data is expected to be used during the execution of the kernel.

The wrapper function should connect the outer and inner interfaces based on its arguments. In our current implementation, it can be expressed via an abstract C program that sets up values before calling the kernel.

4.6.3 Interacting with the Embedding Environment

Besides the considerations of how to generate code for dataflow graphs, we must discuss some of the details on how to supplement the missing parts required for getting a whole binary image. The interface between automatically generated code and user-supplemented parts is the generated header file (in case of C), but we believe that a similar concept can be given for other backends, too.

As we have noted earlier, there may be tasks in the flow that do not contain any kernel, but are responsible for generating input data or consuming output data. Their definition is not part of the high-level program, so we have to rely on the user to write code for them. In that sense, we treat them as open nodes of the network (see Section 3.7). Thus the user has to implement a function satisfying the previously mentioned “outer interface” (see Section 3.5) in the target language.

In addition to that, there must be a controller function. Its purpose is manifold. It controls the life time of the system: before it is started, the necessary initialization routines are run, and after it is finished, the run-time system shuts itself down, possibly terminating the whole binary. The controller accesses the internals of the global configuration therefore it may change the parameters of the programs in the flow network. This way, it may generate events at run time for the constructed system (Figure 4.6). As we have mentioned in Section 3.3, events are not explicit elements, they are merely changes in a subset of the global configuration.

![Figure 4.6: Propagation of information from the controller to the individual kernels via the shared global configuration.](image)

For example, it can be considered the main() function of the application for the C language.

For each target, there shall be a compile function implemented on top of the functions in the PrimKern type class. It generates a series of definitions (i.e. code) for the given
target based on the name of the constructed flow, specified as a regular Haskell `String`. It must also contain an initial global configuration.

```haskell
class Translation a b where
  translateType :: a <- b
```

The `Translation` type class describes a relation between types, stating that there is conversion from type `a` to type `b` if an instance for `Translation a b` exists. It is used to express that the type representing the global configuration (`c`) may be translated to the target language.

```haskell
class Translation a b where
  translateType :: a <- b
```

### 4.6.4 Run-Time Support

Besides the client code (see Subsection 4.6.3), the generated code requires some run-time support to run. These are the basic abstractions required for operating the model presented in the chapter: task pools (selectors), workers, message queues. The implementation of the abstractions depends on the platform where we want to run the system, they are presented in the target language of the backend (that we have called `t` before). Hence the target language must be paired up with a deployment platform.

Semantics for basic abstractions was given earlier (see Section 3.8 for the message queues, Section 4.2 and Section 4.5 for the task pools and workers). The amount of run-time support required on the platform in question is matter of deployment. The software stack under the `Flow` program may consist of a conventional operating system (e.g. a microkernel, a set of servers representing system services or any POSIX-compliant system\(^7\)), and run-time libraries for the target language (Figure 4.7a), or it may be run directly on top of the bare metal (Figure 4.7b).

\(^7\)The larger the underlying system, the less advantage we provide.
4.7 Summary

We have presented a possible implementation of how to run Flow programs in parallel, achieving and maintaining a certain level of efficiency. We abandoned the idea of the regular preemption-based multitasking, instead we described a specialized task management methodology to implement a task-level parallelism for Flows. In this scheme, each of the small programs that we have previously taken as node of a data dependency graph are wrapped further into tasks. Tasks are technically nodes packed with information on their input and output parameters so they are ready to be executed by a worker.

Workers were considered the representation of an execution unit that does not do any preemption but waits for the currently selected task to complete its operation then it looks for another task to be run. That helps to avoid the (sometimes high) costs of context switching and cache thrashing by respecting the lines drawn by the application developer. Certainly, a weakness of the approach may be that it is depending on the programmer, as if one of the tasks stuck for a long or even infinite time then the application sticks together with that as well. However, given that the goal of the constructed system is to map all incoming input data to an output data (like a function), a preemptive scheduling would also stuck due to the data dependencies introduced between the components of the system.

Moreover, managing execution of the Flow graph nodes with atomically-run tasks may be enhanced further by organizing the tasks into pools that are assigned the workers. That way a queue of tasks is constructed, but with two beneficial properties. Task pools may be given explicitly by the user through a set of combinators, that is, the user is capable to hint which sections of the graph may be run efficiently in parallel when executed besides that she has already partitioned the program by data dependencies of the components. Furthermore, it has been also revealed that task pools may not be necessarily real queues but represented with functions.

The functions assigned to task pools were called selectors that modeled simple heuristics to decide which task to pick from the pool in question. We have distinguished two main types of the possible scheduling schemes, where one of them followed the regular round-robin algorithm (that has been also featured in the earlier chapters), meaning that always the first available task is selected, while the another one operates based on a property of the tasks. That latter promises a good dynamic partitioning of the problem as it evaluates the corresponding property at each scheduling step, adapting to the current state of the processing dynamically. As task pools are assigned to each of the workers and scheduling is expressed through selector functions as properties of the pools, scheduling can be taken as it was distributed and each worker can decide for itself. Albeit it has not been worked out in the chapter, it is worth to add that the defined combinators can be also used by an algorithm to generate scheduling for the given application automatically.

Finally, all these concepts are supported by code generation, where only some guidelines were given. The reason is that we aimed to keep the description of the code generation abstract considering that the semantics of the code to be generated has been already specified by the serial and parallel abstract programs. We believe that it is possible generate programs for several target languages, potentially including today’s popular choice, the LLVM intermediate format – as the role of the Backend type class shows the actual bottleneck here is the compilation support of the languages employed in the description of the dataflow network. The design choices discussed above also contributed to a simpler run-time interface that may be implemented with only a few abstractions in mind. Sim-
plicity in case of run-time systems is virtually a requirement because inside the kernels of traditional operating systems, everything has to be self-hosting. That is, there is usually only a restricted version of standard routines are available.
Chapter 5

Flow Programs as Operating Systems

In order to validate the Flow terminology and methodology introduced in the previous chapters, we will model a simple operating system. The constructed model to be demonstrated here is specific to the domain of digital signal processing where our goal is to provide a combined run-time and operating system for orchestration of DSP algorithms. We assume that such algorithms are already formulated in a dedicated domain-specific language, Feldspar (see Chapter 2). The purpose of Feldspar is to capture the essence of digital signal processing computations at a higher level. In an industrial setting, the algorithms are implemented in C or assembly code, and thus their development involves pain-staking manual hand-crafting of code. In addition, the resulting program is typically difficult to maintain, and, in particular, difficult to port to new platforms. Feldspar addresses the problem by expressing the algorithms in a clean, abstract, and hardware-independent way. Initial experiments have shown that it is easier to design and implement algorithms at this level of abstraction compared to that of C or assembly. Feldspar was designed specifically for digital signal processing algorithms, taking into account the specific constructs of the field. It expresses algorithms in a declarative manner instead of using hardware-dependent low-level constructs. A compiler was developed for Feldspar to bridge the gap between an abstract, easy-to-understand source program and the highly optimized target code which makes use of the special features of the digital signal processing hardware. All of those features make Feldspar an ideal choice for our demonstration purposes.

Here we present a possible extension to Feldspar using the Flow language, which can be used to establish additional language constructs and computation primitives that can be used as a high-level description of how to operate Feldspar programs on top of embedded hardware, e.g. Tilera processors. That is to say, in a broader sense, we are describing a way to model operating systems in the Flow language while simultaneously making the implementation details more concrete in the context of the featured example. Domain-specific languages are worked out for a particular problem domain, together with the corresponding program representation and computation model. The same could also be employed in construction of operating systems, thus now we demonstrate how to engineer them towards a well-characterized domain. Domain-specific languages are already used to describe parts of operating systems, or express operating systems as a collection of carefully designed domain-specific languages (c.f. Barrelish [67]). However, it is not common to consider an operating system itself as a matter of specialization (c.f. Mirage [47]). Although there are certain situations where traditional operating systems (e.g. Linux or BSD) are employed in environments that are very specific to certain domains,
they cannot be considered fully optimized systems in general.

Our model represents an alternative and more specialized design that instead tries to exploit the knowledge of the application domain. In the systems we model, several discrete steps of digital signal processing transforms are composed into applications to perform a complete functions. Thereby simultaneously handling multiple flows of data with similar chains of processing configured dynamically and individually. Specifications of such systems are often written in a manner that emphasizes a compositional style, and algorithms are described by themselves and their different configuration-dependent compositions. The majority of such problems consist of designing the interaction of the parts in the processing chain. Properties of the interaction include method of data transfer, data format, and spatial locality.

5.1 Synopsis

In this chapter, we first start with a general characterization of systems employed in the field of digital signal processing. It is followed by a concrete example called audioproc that is slowly extended to cover the use cases from the field. In parallel with that, we describe how to fit Feldspar to the Flow language and thus how to tailor a modelling language for our needs. Up to this point we have not provided much detail on a Flow can be simulated or compiled to a specific target language – now, with the help of the featured audioproc example, we will take the opportunity to walk through those details as well.

Note that, for the sake of clarity and simplicity, we will not reach the level of hardware-dependent specifics during the discussion of the example run-time system and the user-specified extensions. As we noted in the schematic description of the run-time support in the previous chapter, there can be many different deployment scenarios where, for example, a microkernel with some support for the Flow primitives (i.e. workers, tasks, task pools, message queues) is sufficient. In this example Flow program, there will be known open edges (represented as open nodes whose implementation have to be given in the target language). In other words, we rely on the user to feed the dataflow network with and consume its generated output. We believe that such parts can be addressed easily by the practitioners as even operating systems written in C have to contain sections implemented in platform-specific assembly to some degree. Fortunately, such sections are usually small and simple, and their only purpose is to fill in the gaps between the hardware-specific details and platform-independent body of the system. Given that there are experiments to describe hardware interfaces [64] or application binary interfaces [10], it seems likely that such areas will be covered in the future.

5.2 Overview

The purpose of the proposed extension is to describe the structure (relationship) of algorithms implemented in Feldspar and their reactions to various external events (reactive behavior). Our assumption is that the system in question is built upon a data dependency graph, waiting for input and events. The system is continuously running, i.e. processes input while accepts event and handles them. It is also assumed to be a massively parallel system with nearly identical chains of processing.
Feldspar functions are represented as kernels (Section 3.5). The parameters that are passed to the functions are represented as kernel configurations (Section 3.3). There can be several kernels – with or without parameters – composed into a “segment” where kernels can be switched between during operation. There can be pipelines built out of segments, connected by channels. The behavior is described in terms of events and the actions they trigger. Events are entities (e.g. control information) entering the system from the outside world, and which are acted upon by the system. Typical events and their handlers: change a kernel in a segment, update configuration for a kernel.

An overview of a typical digital signal processing system is presented in Figure 5.1. Pipelines are usually clearly segmented for at least two reasons: they may contain parts with different processing speeds, and some parts of the system may be controlled differently. For instance, coding and decoding phases of processing may change algorithm in use based on the quality of the data to be processed. According to this observation, several “control areas” can be formed within the pipelines (as segments), instructed by control messages. It can be modeled as a switching network of different options, where switches are induced by (control) events.

5.3 Sketching Up a Dataflow

Let us suppose that we want to build a simple audio processing application from a set of domain-specific programs written in Feldspar (as shown in Figure 5.2). The application reads digitalized stereo sound signals from the machine’s sound device, applies a given effect to it, then writes the result back to the device thus causing it to emit a transformed version of the sound read previously. Since the computer possibly supports concurrent execution of programs via multi-threading, we plan to parallelize the processing by splitting up the input signal by its left and right channels. Thus we get two identical lines of processing to handle the corresponding channels. We also decompose those lines into smaller segments where the spectrum of the signal is calculated first, followed by the transformation (i.e. the effect), and finally translated back to a waveform.
Figure 5.2: An overview of a sample audio processing application.

As a language focusing solely only on the elements of digital signal processing, *Feldspar* is clearly missing the support for creating standalone applications, for instance, managing resources of the underlying hardware. If we observe Figure 5.2 closely, we may note that while contents of each small box may be described in *Feldspar*, their combinations and the circle-shaped elements may not.

That is where *Flow* can be used as a “glue language” to connect the individual components into an application. As we have learnt above (see Section 3.2), the *Flow* language builds upon combinators to express data dependency graphs that may be taken as dataflow networks. They represent a loop body in an event-driven system that constantly receives input to produce output as an answer. With *Flow*, our goal would be to make the construction of such and similar applications simple if a set of algorithms programmed in one or more domain-specific languages is given. In the optimal case, it would even fit into a few lines of code, like that:

```haskell
audioproc = split --< (processing, processing) >-- merge
  where processing = fft --> effect --> ifft
```

### 5.3.1 Attaching *Feldspar* to the Flow

Let us now assume that the code snippet above is a valid description of the system we want to model. The whole application here is named *audioproc* – it is a top-level Haskell function that is also to represent a *Flow* program. Each small box from Figure 5.2 is added as additional Haskell function to represent a node in the dataflow graph.

*Feldspar* is represented as the *Feldspar* type, as we have done in Section 3.4.

```haskell
data Feldspar = Feldspar
```

However, in order to fit *Feldspar* into a *Flow* program, we have to provide the necessary mappings between *Feldspar* types and *Flow* types (as discussed in Section 3.4). The application will need the support the *Data Float*, *Data (Complex Float)*, and *Data Index* (which is the synonym of *Data DefaultWord*) *Feldspar* types.

```haskell
getDataRep :: (Syntactic a) => a -> DataRep
dataRep = dataRep . eval
```
Here we use the `getDataRep` function that helps to extract *Feldspar* values. The `Syntactic` type class in *Feldspar* characterizes all types that have concrete representation\(^1\) – which are our focus. In the definition above, the `getDataRep` function first evaluates a value with type in the `Syntactic` class.

```haskell
eval :: Syntactic a => a -> Internal a

where actually

instance Type a => Syntactic (Data a)

and

instance Type a => MultiEdge (Data a) Feldspar EdgeSize
    where
        type Internal (Data a) = a
        ...

and

class Type a where
    dataRep :: a -> DataRep
    ...
```

That is, the partial type class (`Type`) and instance (`MultiEdge`) definitions above mean that programs in *Feldspar* are represented as multi-edge graphs. Therefore the evaluation of *Feldspar* expressions are implemented in terms of finding the result as a node of such graph – the `Type a` constraint shows that only types with instances to the `Type` class may be represented in that form. The `Syntactic` is only a specialization of the `MultiEdge` class for *Feldspar* programs.

The `dataRep` function of the `Type` type class is then used to retrieve a concrete (`DataRep`) representation for the given value. Note that it is defined similarly to our `Chunk` type (as presented in Section 3.4).

```haskell
data DataRep
    = BoolData Bool
    | IntData Integer
    | FloatData Float
    | ComplexData DataRep DataRep
    | ArrayData [DataRep]
    | StructData [DataRep]
```

Besides these, we will need the `value` function which represents a simple *Feldspar* program to compute a constant value for each type that is a member of the `Type` type class.

```haskell
value :: Type a => a -> Data a
```

---

\(^1\)That is, they are not virtual, like the vectors presented earlier.
With these tools in our hands, now we are ready to define the required DSLType instances for *Feldspar*.

```haskell
instance DSLType Feldspar (Data DefaultWord) where
type T Feldspar (Data DefaultWord) = Int

fromDSL Feldspar v =
  case (getDataRep v) of
    IntData x -> fromIntegral x

toDSL Feldspar = value . fromIntegral

instance DSLType Feldspar (Data Float) where
type T Feldspar (Data Float) = Float

fromDSL Feldspar v =
  case (getDataRep v) of
    FloatData x -> x

toDSL Feldspar = value

instance DSLType Feldspar (Data (Complex Float)) where
type T Feldspar (Data (Complex Float)) = Complex Float

fromDSL Feldspar v =
  case (getDataRep v) of
    ComplexData (FloatData r) (FloatData i)) -> r :+ i

toDSL Feldspar = value
```

For the *DVector* type there must be a helper type class (*GetData*) introduced to express a polymorphic mapping. Its role is to simplify the definition of the DSLType instances for the complex *Feldspar* types.

```haskell
class GetData a where
  getData :: DataRep -> a

instance GetData Float where getData (FloatData x) = x
instance GetData (Complex Float) where
  getData (ComplexData (FloatData r) (FloatData i)) = r :+ i
```

Thus there is the DSLType instance for arbitrary *DVector* in *Feldspar*. Note that they are mapped to regular Haskell lists.

```haskell
instance DSLType Feldspar (Type a, GetData a, FlowType a, DSLType Feldspar (Data a)) => DSLType Feldspar (DVector a) where
type T Feldspar (DVector a) = [a]

fromDSL l (getDataRep -> ArrayData xs) = map getData xs
toDSL l = vector
```
The expressions that we used in our succinct definition of *audioproc* are not directly *Feldspar* programs. As we have noted in Section 3.5, they must be lifted to be part of the *Flow* network. A specialization of the previously introduced *lift* function needs to be defined.

\[
\text{liftFeld} :: (\text{Liftable Feldspar } a \ b \ c \ C, \\
\text{DSLType Feldspar } a, \text{DSLType Feldspar } b, \text{DSLType Feldspar } c) \\
\rightarrow (c \rightarrow a \rightarrow b) \rightarrow \text{Flow } (T \text{ Feldspar } a) (T \text{ Feldspar } b) (T \text{ Feldspar } c) C
\]

\[
liftFeld = \text{lift Feldspar}
\]

In addition to this, *Feldspar* has an associated helper data type (*FK*) implemented as an instance of *PrimKern*. This is due to the *Compilable* type class found in the *Feldspar* C compiler. As its name suggests, that class contains the technical functions required to generate code for the given *Feldspar* program. Every instance of the *Compilable* class must be also member of the *Syntactic* type class.

\[
\text{instance Syntactic } a \Rightarrow \text{Compilable } a \text{ where}
\]

\[
\ldots
\]

We must handle the regular, splitter, and merger type functions separately. Here they are denoted by the K0, K1, and K2 constructors, respectively.

\[
\text{data FK} :: * \rightarrow * \rightarrow * \rightarrow * \rightarrow * \rightarrow * \text{ where}
\]

\[
\text{K0} :: (\text{Syntactic } c, \text{Syntactic } a, \text{Syntactic } b, \text{Compilable } (c \rightarrow a \rightarrow b)) \\
\rightarrow (c \rightarrow a \rightarrow b) \rightarrow \text{FK } a \ b \ c \ \text{Feldspar } C
\]

\[
\text{K1} :: (\text{Syntactic } c, \text{Syntactic } a, \text{Syntactic } b0, \text{Syntactic } b1, \text{Compilable } (c \rightarrow a \rightarrow (b0,b1))) \\
\rightarrow (c \rightarrow a \rightarrow (b0,b1)) \rightarrow \text{FK } a \ (b0,b1) \ c \ \text{Feldspar } C
\]

\[
\text{K2} :: (\text{Syntactic } c, \text{Syntactic } a0, \text{Syntactic } a1, \text{Syntactic } b, \text{Compilable } (c \rightarrow (a0,a1) \rightarrow b)) \\
\rightarrow (c \rightarrow (a0,a1) \rightarrow b) \rightarrow \text{FK } (a0,a1) \ b \ c \ \text{Feldspar } C
\]

Then the required *liftPK* instances are given as follows.

\[
\text{instance (PrimKern FK Feldspar } C, \text{Syntactic } a, \text{Syntactic } b, \text{Syntactic } c) \\
\Rightarrow \text{Liftable Feldspar } a \ b \ c \ C \text{ where}
\]

\[
liftPK \text{ Feldspar id} = \text{Kern id . K0}
\]

\[
\text{instance (PrimKern FK Feldspar } C, \text{Syntactic } a, \text{Syntactic } b0, \text{Syntactic } b1, \text{Syntactic } c) \Rightarrow \text{Liftable Feldspar } (b0,b1) \ c \ C \text{ where}
\]

\[
liftPK \text{ Feldspar id} = \text{Kern id . K1}
\]

\[
\text{instance (PrimKern FK Feldspar } C, \text{Syntactic } a0, \text{Syntactic } a1, \text{Syntactic } b, \text{Syntactic } c) \Rightarrow \text{Liftable Feldspar } (a0,a1) \ b \ c \ C \text{ where}
\]

\[
liftPK \text{ Feldspar id} = \text{Kern id . K2}
\]

By the aid of the type class instances and definitions above, there may be an instance\(^2\) given for the FK type, and so for *Feldspar*.

\(^2\)Yet only partial instance, though.
instance PrimKern FK Feldspar C where
  run (K0 f) cfg x = fit Feldspar f cfg x
  run (K1 f) cfg x = fit Feldspar f cfg x
  run (K2 f) cfg x = fit Feldspar f cfg x

where the definition of the fit function is given as follows.

fit l f c = fromDSL l . f (toDSL l c) . toDSL l

5.3.2 Anatomy of the High-Level Description

In this application, split reads the input, splits it into two, and passes each part to an instance of the audio processing line, described by processing. The actual implementation of the Feldspar function behind split (splitFeld) is shown in Figure 5.3: because stereo signals store data interleaved by channels, the splitting is performed by moving every odd and every even element of the input vector into two separate subvectors as a result.

split --< (processing, processing)

splitFeld :: DVector Float -> (DVector Float, DVector Float)
splitFeld v = (v1,v2)
    where
      h = length v 'div' 2
      v1 = indexed h (\idx -> v ! (2 * idx)
      v2 = indexed h (\idx -> v ! (1 + 2 * idx))

Figure 5.3: The definition of split in Feldspar.

The results are then combined by using the merge function that uses the mergeFeld Feldspar program (see Figure 5.4): which is the exact opposite of splitting, that is, the elements of the input vectors are interleaved to produce the resulting combined vector.

(processing, processing) >-- merge

mergeFeld :: (DVector Float, DVector Float) -> DVector Float
mergeFeld (v2,v1) = indexed (length v1 + length v2) f
    where f i = (i 'mod' 2 == 0) ? (v1 ! ix, v2 ! ix)
        where ix = i 'div' 2

Figure 5.4: The definition of merge in Feldspar.
Steps in processing are connected by the \(\rightarrow\) combinator. It is used to describe a sequence of computational stages. For example, a Fast Fourier Transform followed by an effect and an inverse FFT. Note that independent of the way we sequentially connected the components here, they are going to run in parallel. This is because communication between each of them will be implemented in an asynchronous way (see Subsection 5.6).

\[
\text{processing} = \text{fft} \rightarrow \text{effect} \rightarrow \text{ifft}
\]

\[
\text{fftFeld} :: \text{DVector Float} \rightarrow \text{DVector (Complex Float)}
\text{fftFeld} = \text{M.fft} \cdot \text{map} (\lambda v \rightarrow \text{complex} v 0.0)
\]

Figure 5.5: The definition of fft in Feldspar.

\[
\text{ifftFeld} :: \text{DVector (Complex Float)} \rightarrow \text{DVector Float}
\text{ifftFeld} = \text{map realPart} \cdot \text{M.ifft}
\]

Figure 5.6: The definition of ifft in Feldspar.

The fft and ifft functions are derived from the fftFeld (see Figure 5.5) and ifftFeld (see Figure 5.6) Feldspar programs. Note that we omitted the definition of the M.fft and M.ifft functions as they are released as part of the standard Feldspar releases (e.g. [17]). In the functions we simply map the Data Float (real) elements to Data (Complex Float) (complex) ones.

The effect function may be a bandpass filter (bandpassFeld), an octaver filter (octaveUpFeld), or an identity filter (noneFeld) that does no transformation. They are shown in Figure 5.7.

### 5.3.3 Introducing Dynamism

In order to illustrate the ability to model individually controlled segments in the pipelines, we will implement a per-channel switching between the effects at run time (see Figure 5.8). As we mentioned previously it is a common requirement for systems that we want model here: they have to process data coming from multiple sources in a similar way. Although there may be settings that vary piece by piece. For example, the system in question may function as a radio receiver that digitizes and forwards signals over a network link while maintaining connections with mobile device via radio waves. The quality of the radio link of each client strongly depends on the strength of the signal that our system – as a radio station – has to follow. The system may vary the way the signals are decoded based on how much information it receives.

Introducing such dynamism does not require a great deal of modification. We are not working with a radio station at the moment, but a similar change can be implemented here too. Only the effect has to be factored out from processing and extended to support switching between effects. As one can see this is implemented at the level of the domain-specific language, i.e. in Feldspar, by the Switch type class.

\[
\text{class Switch s a b | s} \rightarrow \text{a, s} \rightarrow \text{b where}
\]

\[
\text{switch :: Data Index} \rightarrow \text{s} \rightarrow \text{a} \rightarrow \text{b}
\]
bandpassFeld
:: (Data Index, Data Index) -> DVector (Complex Float)
-> DVector (Complex Float)
bandpassFeld (x,y) = filterIndex (\v -> x <= && v <= y) 0

filterIndex :: (Type a)
=> (Data Index -> Data Bool) -> Data a -> DVector a -> DVector a
filterIndex cond elemIfFalse v =
    indexed (length v) (\i -> (condition (cond i) (v ! i) elemIfFalse))

octaveUpFeld :: DVector (Complex Float) -> DVector (Complex Float)
octaveUpFeld input = indexed n (\i -> condition (cond i) (f i) 0)
where
    n = length input
    cond i = i `mod` 2 == 0
    f i = input ! (i `div` 2)

noneFeld :: DVector (Complex Float) -> DVector (Complex Float)
oneFeld = id

Figure 5.7: The definition of filters bandpass, octaveUp, none in Feldspar.

The purpose of this type class is to provide overloading for tuples of different sizes, representing many-way switches. For example, 2- and 3-way switches may be given as follows.

instance (Syntactic a, Syntactic b)
=> Switch (a -> b, a -> b) a b where
    switch e (f1,f2) input = (e == 0) ? (f1 input, f2 input)

instance (Syntactic a, Syntactic b)
=> Switch (a -> b, a -> b, a -> b) a b where
    switch e (f1,f2,f3) input =
        (e == 0) ? (f1 input, (e == 1) ? (f2 input, f3 input))

Note that we are using functional dependencies [27] in the implementation of Switch to express the relationship between the functions to be switched between. They must have the same type, i.e. the same interface, otherwise we could not ensure the static correctness of the graph.

Apart from the actual indices of the effects activated for the individual channels, we can also spot that the bandpass filter has two additional run-time parameters. The scheme then can be used to add switching to both pipelines.

effectFeld :: Data Index -> (Data Index, Data Index)
-> DVector (Complex Float) -> DVector (Complex Float)
effectFeld e bpp = switch e (noneFeld,bandpassFeld bpp,octaveUpFeld)

As a result, we can give a type for the global configuration as well.
Figure 5.8: An overview of the audio processing application, extended with dynamism.

type Config = (Data Index, Data Index, Data Index, Data Index)

Let us say that the first two elements of the 4-tuple be the indices of the active filters for the left and right channels, respectively, while let the latter two elements be the corresponding run-time parameters of the bandpass filtering.

Thus, finally we can give the complete definitions for liftings. Note that for *Feldspar* programs that do not have parameters, we ignore the global configuration.

\begin{align*}
\text{split} &= \text{liftFeld} (\_ \rightarrow \text{splitFeld}) \\
\text{merge} &= \text{liftFeld} (\_ \rightarrow \text{mergeFeld}) \\
\text{fft} &= \text{liftFeld} (\_ \rightarrow \text{fftFeld}) \\
\text{ifft} &= \text{liftFeld} (\_ \rightarrow \text{ifftFeld}) \\
\text{effectLeft} &= \text{liftFeld} ((e,\_ \rightarrow \text{fftFeld}) \rightarrow \\
& \quad \text{effectFeld e (bp1,bp2)}) \\
\text{effectRight} &= \text{liftFeld} ((\_ \rightarrow \text{fftFeld}) \rightarrow \\
& \quad \text{effectFeld e (bp1,bp2)})
\end{align*}

Because the effect had to be split by channels, we will need to rewrite the processing lines for two channels as well.

\begin{align*}
\text{processLeft} &= \text{fft} \rightarrow \text{effectLeft} \rightarrow \text{ifft} \\
\text{processRight} &= \text{fft} \rightarrow \text{effectRight} \rightarrow \text{ifft}
\end{align*}

Finally, there is the improved version of the top-level expression.

\begin{align*}
\text{audioproc'} &= \text{split} \rightarrow (\text{processLeft}, \text{processRight}) \rightarrow \text{merge}
\end{align*}
5.3.4 Marking Kernels

The programmer may choose to describe the application differently where the segments are “fused” in Feldspar, that is, all the intermediate data structures are removed as result of the compilation. As a consequence of the fact that every kernel is run to completion, it is guaranteed that the complete contents of each processing line will be run on the same processor without interruption as they are now considered a single kernel in the dataflow network. And, since there is no communication between different processing units needed, the necessary serialization and deserialization of data may be optimized away.

\[
\text{processFeld} :: \text{Data Index} \rightarrow (\text{Data Index, Data Index}) \\
\rightarrow \text{DVector Float} \rightarrow \text{DVector Float}
\]

\[
\text{processFeld} e \text{ bpp} = \text{ifftFeld} . \text{effectFeld} e \text{ bpp} . \text{fftFeld}
\]

In this case, the application is written as follows.

\[
\begin{align*}
\text{left} & = \text{liftFeld} (((e,\_,bp1,bp2) :: \text{Config}) \rightarrow \text{processFeld} e (bp1,bp2) \\
\text{right} & = \text{liftFeld} (((\_,e,bp1,bp2) :: \text{Config}) \rightarrow \text{processFeld} e (bp1,bp2)
\end{align*}
\]

\[
\text{audioproc''} = \text{split} --< (\text{left, right}) >-- \text{merge}
\]

Fusing FFT and inverse FFT to the effects also offers the opportunity to optimize the case when no effect has to be computed, i.e. there is no change to the signal effectively. In the previous descriptions, both of FFT and its inverse will be computed regardless of whether there is an effect applied or not. It may be also beneficial if we add effects to the application that do not require the Fourier transform of the signal (e.g. overdrive).

\[
\text{processFeld'} :: \text{Data Index} \rightarrow (\text{Data Index, Data Index}) \\
\rightarrow \text{DVector Float} \rightarrow \text{DVector Float}
\]

\[
\text{processFeld'} e \text{ bpp} = \\
(\text{e} == 0) ? (\text{input}, (\text{ifftFeld} . \text{effectFeld} e \text{ bpp} . \text{fftFeld}) \text{ input})
\]

5.3.5 Scheduling

By using the concepts we introduced earlier for declarative scheduling (see Section 4.5), audioproc may be tagged in the following way to improve its performance.

\[
\begin{align*}
\text{pool1} & = \text{createPool} 1 \\
\text{pool2} & = \text{createPool} 2
\end{align*}
\]

\[
\text{audioproc} = \text{split} --< (\text{processLeft, processRight}) >-- \text{merge}
\text{where}
\begin{align*}
\text{processLeft} & = \text{pool1} #= \text{fft} --\text{effectLeft} --\text{ifft} \\
\text{processRight} & = \text{pool2} #= \text{fft} --\text{effectRight} --\text{ifft}
\end{align*}
\]

This way three independent task pools are created for the Flow program, grouping only the kernels that may be more or less executed independently of each other. Sub-flows tagged with the same identifiers become the part of the same pool.

It may be improved further with the addition of a simple dynamic scheduler. That scheduler is to pick a task with the longest queue. We use the definition of \text{maxQLen} from the Section 4.5.
pool = scheduleBy maxQLen (usePool 0)

audioproc = pool #=
    split --< (processLeft, processRight) >-- merge
where
    processLeft  = fft --> effectLeft  --> ifft
    processRight = fft --> effectRight --> ifft

5.4 Simulating the Dataflow

Note however, that the circle-shaped nodes of Figure 5.8 are completely missing from the resulted code. This is because they are the implicit source and sink nodes for the graph. The input enters the application through the source node, and the output is sent to the sink node. As we have mentioned previously, they are treated as open nodes of the graph. Such nodes do not have concrete programs associated in this high-level view, though they may be substituted by suitable producer and consumer functions in Haskell.

Substituting open nodes in the graph becomes useful when we would like to see how the application is working, i.e. when we want to simulate the behavior of the application. The audioproc is now represented as a Flow program that may be run and analyzed directly in a Haskell interpreter (e.g. GHCi for the Glasgow Haskell Compiler) by the using the previously introduced simulate and eject functions (see Section 3.8).

The simulate function performs the computation represented by audioproc endlessly as such a system never stops. Hence we pick only one of the generated states and use the eject function to access to output value in that step.

As we have shown, by introducing a global state, we have created a way to directly affect the behavior of the flow network at run time by modifying the elements of the global state. Note that programs in the Flow program cannot modify the global state themselves, that can be changed from the outside only. Hence the global configuration is going to specified as an infinite list of states to be consumed at each step of the simulation. Elements of that list represent actual states of the configuration at given moments.

Similar to the data passed between the wrapped programs, the global configuration is also expressed in the common data types because it must be independent of the employed domain-specific language. For the audioproc application it can be given as follows.

*Audioproc> let audioprocConfig = (1::Int,1::Int,0::Int,100::Int)*
*Audioproc> let config = repeat audioprocConfig

where the corresponding elements of the Haskell tuple refer to each of the initial states of the effects for the channels, and the parameters for the bandpass effect. Note that global configuration here is not changing over time.

*Audioproc> let input = [[1..32 :: Float], [1..32]]
*Audioproc> eject (simulate audioproc' input config !! 31)
[[9.000001,10.000001,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,
20.0,21.0,22.0,23.0,24.0,9.000001,10.000001,11.0,12.0,13.0,14.0,
15.0,16.0,17.0,18.0,19.0,20.0,21.0,22.0,23.0,24.0], [9.000001,
10.000001,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0,21.0,
22.0,23.0,24.0,9.000001,10.000001,11.0,12.0,13.0,14.0,15.0,16.0,17.0,
18.0,19.0,20.0,21.0,22.0,23.0,24.0]]
If we run the application with a different configuration then we get different results for the same input, while the same input values imply the same output values, establishing a pure model.

*Audioproc> let audioprocConfig = (2::Int,2::Int,100::Int,200::Int)
*Audioproc> let config = repeat audioprocConfig
*Audioproc> eject (simulate audioproc' input config !! 31)

[[1.0,2.0,3.000001,4.000001,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,
  14.0,14.999998,15.999998,17.0,18.0,19.0,20.0,21.0,22.0,23.0,24.0,
  25.0,26.0,27.0,28.0,29.0,30.0,31.000002,32.0],[1.0,2.0,3.000001,
  4.000001,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,14.999998,
  15.999998,17.0,18.0,19.0,20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,
  28.0,29.0,30.0,31.000002,32.0]]

5.5 Compiling the Dataflow to C

In order to generate C code for the application, we will need to continue the definition of the PrimKern instance (from Section 3.5). First, we will sketch up the missing definitions of compile and finalize from the PrimKern type class.

instance PrimKern FK Feldspar C where
  compile k ix = (decls, name, body)
  where
    name = "primkern_" ++ show ix
    decls = toCTypes dcs
    (dcs,body) = case k of
      K0 f -> compileK0 f
      K1 f -> compileK1 f
      K2 f -> compileK2 f

  finalize _ = replace "%%FUNCTION%%"

The compileK0, compileK1, compileK2 functions are essentially specializations of the same function for the types of function stored with the constructors K0, K1, K2, respectively. This function is the one that is responsible for compiling Feldspar programs to C. It generates a tuple with the following information:

- **Declarations.** Abstract representation of the types required for compiling the resulted program. Because each Feldspar program is compiled individually when the code is generated for the application, it may happen that some programs will require the same type (e.g. struct) definitions. They are initially given as elements of the Abstract Imperative Representation of Feldspar so they must be mapped to the elements of CTypes. This is implemented as an instance of Translation (Section 4.6).

---

3 We used the replace function from the MissingH package: http://hackage.haskell.org/package/MissingH.
Program body. The program body itself is already a string, thus there is no further conversion needed. However, the Feldspar program is initially compiled with the name %FUNCTION%. It has to be able to generate a function body without a name first (for the first phase of the code generation), then to be used with the finalize function to insert the name for the function (if that has not been eliminated). This explains our choice for the name: %FUNCTION% is not a valid identifier in C, so when we replace it later on, we will replace the “name” of the function.  

Second, a task wrapper function (see Section 3.5) for Feldspar programs must be defined. As we discussed earlier, the purpose of the wrapper is to connect the inner interface of the translated Feldspar program with the outer interface for tasks. That is where we can fill in the hole left between Feldspar and Flow.

A task wrapper function is implemented in terms of an abstract version of the C language that contains similar constructs to the original C language but as a simple embedded language in Haskell, as part of the C backend support of the Flow language. We believe that, the fact that we are still able to use Haskell here as meta-language, even being able to define higher-order functions to describe code templates with types, is a clear advantage. However, the wrapper can only work if it receives the information about the heap allocations required for running the given Feldspar program.

This is where the definition of the heapInfo function has to be unfolded. Here we call the info function of Feldspar that tells what type of data is stored outside of the stack of the C function that the program has been translated to.  

\[
\text{info} :: (\text{Compilable } t) \Rightarrow t \rightarrow ([\text{Type}], [\text{Type}], [\text{Type}])
\]

We only need the first element from the triplet as that contains information on the local allocations of the function.

\[
\text{instance PrimKern FK Feldspar C where}
\]

\[
\text{heapInfo (K0 } f) = (\text{toCTypes } . \text{fst3 } . \text{info} ) f
\]

\[
\text{heapInfo (K1 } f) = (\text{toCTypes } . \text{fst3 } . \text{info} ) f
\]

\[
\text{heapInfo (K2 } f) = (\text{toCTypes } . \text{fst3 } . \text{info} ) f
\]

\[
\text{fst3 :: (a,b,c) \rightarrow a}
\]

\[
\text{fst3 } (x,\_,\_) = x
\]

Basically, the Feldspar C compiler delegates all of its memory allocations to an upper layer that has more knowledge of the embedding environment. The stack is usually very small so temporary vectors used for some algorithms cannot be placed there conveniently. Only simple variables are allocated on the stack, all other objects are collected during the compilation and added to a special C struct called array as of the current version of Feldspar.

---

4We assume that C code generation for Feldspar programs will never add that string in to bodies anywhere else. The possibility can be minimized by replacing only the first instance that is at the beginning of the function as the name of the function, preceded only by the type of the function.

5The info function for Feldspar is not yet part of the public release as of the time of writing, but it is included in the development version.

6It has the size of about 4 kilobytes.
struct array {
    void *   buffer;
    unsigned int length;
    int       elemSize;
};

The *buffer* field holds a pointer to a memory region, allocated with the size stored by the *length* field, and with elements of size of *elemSize*. Note that the value of the *elemSize* field may be -1 that means there is another *struct array* stored inside. This makes is possible to nest such *arrays* and pack multiple allocated *arrays* into a single allocation.

Such an *array* becomes a constant parameter – called *mem* – for every compiled *Feldspar* function that must be populated before running it. So the wrapper for the *Feldspar* function has to forward the next elements from the input queues to the function and store its results in the output queues. Before running the function, its *mem* parameter has to be initialized according to the information given by *heapInfo*. The initialization is essentially just setting the proper length and size information for each requested *array* in conjunction with setting the memory pointer to the appropriate position in the worker’s memory.

A sample task wrapper code is presented below for the *split* kernel. As one can see, there is no heap needed for running the *primkern_split()* function that is the translation of the *splitFeld* *Feldspar* program. Note that *arrays* are also used to model vectors in C.

```c
void f_split(void* m, struct configuration c, 
        float* inp0, size_t insz0, size_t *inr0, 
        float* outp1, size_t outsz1, size_t* outw1, 
        float* outp2, size_t outsz2, size_t* outw2) {
    struct array mem;
    struct array v0;
    struct array v1;
    struct array v2;
    mem.buffer = NULL;
    mem.length = 0;
    mem.elemSize = 0;
    v0.buffer = inp0;
    v0.length = insz0;
    v0.elemSize = sizeof(float);
    *inr0 = v0.length;
    v1.buffer = outp1;
    v1.length = outsz1;
    v1.elemSize = sizeof(float);
    v2.buffer = outp2;
    v2.length = outsz2;
    v2.elemSize = sizeof(float);
    primkern_split(mem, c.cf_0, c.cf_1, c.cf_2, c.cf_3, v0, &v1, &v2);
    *outw1 = v1.length * v1.elemSize;
    *outw2 = v2.length * v2.elemSize;
}
```
For each task’s C function, there is a void pointer with the address of the memory assigned to the worker passed with the size of the maximal allocation possible for the given worker (Section 4.4). For calculating the concrete value of the maximal allocation, a simple unrolled maximum search with the possible memory requirements for each task generated as a C function to be run at the initialization of the application. However, as it is statically computable by the C compiler, it may be optimized away, depending on the compiler.

5.5.1 Added User Code

Note that the definitions presented above are part of the “binding” for the given domain-specific language, they must be written by the author of the language, or by somebody who wants to use the given language with Flow. For a casual user (i.e. the application programmer), only the definition of audioproc and its configuration (the Config type) must be defined, everything else will be generated automatically.

Specifically, the following C header will be generated for audioproc.

```c
#ifndef __AUDIOPROC_H__
#define __AUDIOPROC_H__

struct configuration {
    char cf_0;
    char cf_1;
    char cf_2;
    char cf_3;
};

extern struct configuration config;

int controller(int arg, char** argv);

void f_0(struct configuration* config, float* b0, size_t b0_insz,
         size_t* b0_ousz);

void f_1(struct configuration* config, float* b0, size_t b0_insz,
         size_t* b0_ousz);

#endif // __AUDIOPROC_H__
```

The struct configuration contains the global run-time parameters for the application. It can be seen that the elements of the Config type have been mapped to single char types in C. As we have noted before, the order of the elements in the declaration matches the order of their enumeration in the source n-tuple. Preserving that ordering is deemed to be important as we want to provide a way for the user to interact with the generated code. It is the responsibility of the user to feed the network with data, consume its output and optionally change the elements of the global configuration to regulate the behavior of the flow.
void f_0(struct configuration* config, float* b0, size_t b0_insz, size_t* b0_ousz)
{
    int i, e;
    signed short shortbuffer[b0_insz];

    *b0_ousz = 0;

    if (reader)
        if (pa_simple_read(reader, shortbuffer, b0_insz * sizeof(signed short), &e) >= 0)
            for (i = 0; i < b0_insz; i++)
                b0[i] = ((float)((double)(shortbuffer[i]) / (double)(SHRT_MAX)));

    *b0_ousz = b0_insz;
}

void f_1(struct configuration* config, float* b0, size_t b0_insz, size_t* b0_ousz)
{
    int i, e;
    signed short shortbuffer[b0_insz];

    *b0_ousz = 0;
    for (i = 0; i < b0_insz; i++)
        shortbuffer[i] = ((signed short)(((double)(b0[i])) * (double)(SHRT_MAX)));

    if (writer)
        if (pa_simple_write(writer, shortbuffer, b0_insz * sizeof(signed short), &e) >= 0)
            *b0_ousz = b0_insz;
}

Figure 5.9: A possible implementation of the f_0() and f_1() functions for audioproc.

A sample implementation for the missing functions is presented in Figure 5.10 (controller()) and Figure 5.9 (f_0() and f_1()). There we used the PulseAudio API [61] to access the sound card. It has a simple text-based user interface where the user can switch between different sound effects.

A sample run for the application is as follows.

$ ./audioproc
s: switch effect, p: change parameters, q: quit

Valid values for 's':

0: bandpass
1: octave up
2: null
3: null left
4: null right
5: reset bandpass

s 2
s 1
q
Finished.
int controller(int argc, char** argv)
{
    char cmd[100];
    int e, x, y;

    const pa_sample_spec sample_spec = {
        .format = PA_SAMPLE_S16LE,
        .rate = 44100,
        .channels = 2
    };

    reader = pa_simple_new(NULL, "audioproc", PA_STREAM_RECORD, NULL,
                           "audio source", &sample_spec, NULL, NULL, &e);
    writer = pa_simple_new(NULL, "audioproc", PA_STREAM_PLAYBACK, NULL,
                           "audio sink", &sample_spec, NULL, NULL, &e);

    printf("s: switch, p: change parameters, q: quit\n");
    printf("nValid values for 's':\n");
    printf("0: bandpass\n1: octave up\n2: null\n");
    printf("3: null left\n4: null right\n5: reset bandpass\n\n");

    do {
        scanf("%s", cmd);
        if (!strncmp(cmd, "s", 2)) {
            scanf("%d", &e);
            switch (e) {
                case 0: config.cf_0 = config.cf_1 = 1; break;
                case 1: config.cf_0 = config.cf_1 = 2; break;
                case 2: config.cf_0 = config.cf_1 = 0; break;
                case 3: config.cf_0 = 0; break;
                case 4: config.cf_1 = 0; break;
                case 5: config.cf_2 = config.cf_3 = 0; break;
                default: break;
            }
        } else
            if (!strncmp(cmd, "p", 2)) {
                x = 0, y = 0;
                scanf("%d %d", &x, &y);
                config.cf_2 = x;
                config.cf_3 = y;
            }
    } while (strncmp(cmd, "q", 2));

    printf("Finished.\n");

    if (reader) { pa_simple_free(reader); reader = NULL; }
    if (writer) { pa_simple_free(writer); writer = NULL; }

    return EXIT_SUCCESS;
}

Figure 5.10: A possible implementation of the controller() function for audioproc.
5.5.2 Run-Time Support

Lists of tasks, task pools, and message queues are considered parts of the run-time system, and they have to be represented in C to support the execution of audioproc. Here we briefly present a sample C API for any POSIX-compatible operating system. 

\begin{verbatim}
struct worker_t;
struct taskp_t;
struct configuration;
struct taskinfo_t;

typedef void (*task_ptr)(void *, struct configuration *);
\end{verbatim}

The concepts are captured by the following types: \texttt{worker\_t} is a worker, \texttt{taskp\_t} is a task pool, an \texttt{configuration} is the global configuration. The \texttt{task\_ptr} is only a technical type which is a function pointer. Tasks in the array are stored in a \texttt{taskinfo\_t} structure that contains information on each of them, e.g. address the task body or the addresses of the associated queues.

\begin{verbatim}
struct worker_t *
worker_create(size_t, struct taskp_t *, struct taskinfo_t *,
             struct configuration *, int);
void worker_destroy(struct worker_t *);
void worker_start(struct worker_t *);
void worker_stop(struct worker_t *);
\end{verbatim}

We can create and destroy workers, and we can also start or stop them after creation. It helps to follow the standard thread pooling pattern, where first we create all workers and then start them. The reverse happens at shutdown. In addition to that, it can be also specified how much memory is given to the worker to work with, and a value for suspending the execution when it cannot find a task to run.

\begin{verbatim}
struct msgq_t;

struct msgq_t * msgq_create(size_t);
void    msgq_destroy(struct msgq_t *);
size_t  msgq_send(struct msgq_t *, void *, size_t);
size_t  msgq_recv(struct msgq_t *, void *, size_t);
int     msgq_empty(struct msgq_t *, size_t);
int     msgq_full(struct msgq_t *, size_t);
\end{verbatim}

There can be message queues created and destroyed as well. We can send and receive data over the queues that represent the channels. The message queues work with bytes without types. In other words, we do not use typed channels in C as the upper layer (the language frontend) has already checked for the proper connections so no further type checking is needed. The direct translation of the \texttt{Chunk} type (see Section 3.4) shares the same purpose.

\footnote{In our opinion, this requirement is not a serious restriction as many embedded systems feature operating systems, e.g. with the Linux kernel, with a POSIX-compliant C API where basic management of memory and threads are implemented. However, it is not a major undertaking to create a run-time system dependent on no other systems as our proposed API is quite minimal.}
#define NO_TASK_FOUND (-1)

typedef int task_t;
typedef task_t (*task_selector)(struct taskp_t *);
typedef int (*task_property)(task_t);

struct taskp_t * taskp_create(size_t, task_selector, task_property);
void taskp_destroy(struct taskp_t *);
void taskp_add(struct taskp_t *, task_t);
task_t taskp_take(struct taskp_t *);
void taskp_drop(struct taskp_t *, task_t);

Task index is represented as a regular C integer. Just as with previous abstractions, we can create and destroy task pools, and put the indices available to them by using taskp_add() as well as get the next appropriate free index by calling taskp_take(). When there is no task to be chosen, a special value, NO_TASK_FOUND is returned. After the execution, the taskp_drop() function is called to return the given task to the pool.

When creating task pools, a selector (as task_selector, an type synonym for a function pointer) and a property (as task_property, a function pointer as well) optionally used for task selection is passed together with the size. The selector must be a valid pointer, while the property may be NULL.

int task_selector_next(struct taskp_t *);
int task_selector_by(struct taskp_t *);

Selectors receive the task pool to be handled. For the task_selector_by() selector, it can access the configured property (passed at the invocation if task_create()) because it is stored as a meta information for the given pool.

int task_property_qlen(task_t);

Properties receive the task index only, so they can get all the task-related information (via the taskinfo_t structure) from the task array. Here a sample definition for the queue length property is given, and further similar ones may be added to that.

As mentioned earlier, task pools may be also represented as queues where the next task to be selected is on the top, so a “put” operation automatically picks that. For the Next selector that would correspond to a FIFO queue, and for the By selector that would be a priority queue where priorities are calculated by the given property or a multi-level feedback queue where the tasks moved between the levels in reflection to the changes in the scores. The implementation of the latter would be complicated as the priority queue has to be reordered constantly as the scores generated are changing, or multi-level queues have to be maintained in a similar fashion. For that very reason, we found it easier to implement the queues with functions. If the number of elements in the pools is kept low, function-represented queues may be an equally good decision.

void * mem_allocate(size_t);
void mem_release(void *);
void suspend(int);

There are some utility functions provided for allocating and releasing memory, as well as suspending the execution of a given thread.
5.5.3 The main() Function

Finally a main() function with the following structure is generated for the application. The function uses some global variables to maintain a state for the message queues (queue, with the length of QUEUE_COUNT which is equal to the number of channels in the abstract program), tasks (task, with the length of TASK_COUNT which is equal to the number of the tasks in the abstract program), a task pool array (task_pool, with a size corresponding to the number of actual pools (TASK_POOL_COUNT) in the application), and a global configuration (config).

```c
struct msgq_t* queue[QUEUE_COUNT];
struct taskinfo_t task[TASK_COUNT];
struct taskp_t* task_pool[TASK_POOL_COUNT];
struct configuration config;
```

The main() function. Note that a “meta-variable” ##n is used to denote sections that shall be unrolled when the actual code is generated.

```c
int main(int argc, char** argv)
{
    struct worker_t* worker[TASK_POOL_COUNT];
    size_t m;
    int c, i;
    /*
     * Initialization of the global configuration with values taken from
     * an initial state (##initialConfig) given at code generation.
     * To be unrolled with ##n = 0 .. (CONFIG_PARAMS - 1).
     */
    config.cf_##n = ##initialConfig##n;
    /*
     * Create the message queues used in the application with sizes
     * corresponding to the actual type of the data moving on the
     * channel (where ELEM##n_SIZE is the size of the type in C, and
     * CHUNK##n_SIZE is the number of elements moving at the same time).
     * To be unrolled with ##n = 0 .. (QUEUE_COUNT - 1).
     */
    queue[##n] = msgq_create(CHUNK##n_SIZE*ELEM##n_SIZE);
    /*
     * Initialization of the task array, where each function
     * representing a task is registered together with the input and
     * output queues connected to them. This line will be unrolled
     * with ##n = 0 .. (TASK_COUNT - 1) and ##mi = 0 ..
     * (NUM_INQUEUES(##n) - 1) and ##mo = 0 .. (NUM_OUTQUEUES(##n) -1)
     * on code generation.
     */
    task[##n].ta.proc = task_##n;
    task[##n].ta.num_inqueues = ##task##n_InQueuesCount;
    task[##n].ta_inqueue[##mi] = queue[##task##n_InQueue##mi];
}
task[##n].ta_num_ouqueues = ##task##n_OutQueuesCount;
task[##n].ta_ouqueue[##mo] = queue[##task##n_OutQueue##mo];
/
* Creation of task pools, to be unrolled with ##n = 0 ..
* (TASK_POOL_COUNT - 1). NUM_TASKS() specifies the number of the
* tasks in the pool, TASK_SELECTOR() gives the active selector, and
* TASKPROPERTY() adds the property used optionally used for
* scoring for ##n, respectively.
*/
task_pool[##n] = taskp_create(NUM_TASKS(##n), TASK_SELECTOR(##n),
   TASK_PROPERTY(##n));
/
* Fill up the task pools with tasks to be executed. To be unrolled
* with ##n = 0 .. (TASK_COUNT - 1), and TASK_POOL_OF() tells which
* pool is assigned to the given task.
*/
taskp_add(task_pool[TASK_POOL_OF(##n)], (task_t)(##n));
/
* The calc_max_memory() is to statically determine how much memory
* is needed for the workers to run any task from the task pool.
*/
m = calc_max_memory();
/
* Create the workers with the number of TASK_POOL_COUNT. That is,
* in this configuration, a single worker is created for each task
* pool.
*/
for (i = 0; i < TASK_POOL_COUNT; i += 1)
   worker[i] = worker_create(m, task_pool[i], tasks, &config, 1000);
/
* Launch the workers at once.
*/
for (i = 0; i < TASK_POOL_COUNT; i += 1)
   worker_start(worker[i]);
/
* Call the user-defined controller function.
*/
c = controller(argc, argv);
/
* After controller() is finished, tear down the stack by stopping
* the workers first.
*/
for (i = 0; i < TASK_POOL_COUNT; i += 1)
   worker_stop(worker[i]);
/
* Suspend the execution for 10 seconds to let all workers to
* finish.
*/
suspend(10000);
/*
 * Destroy workers together with the associated task pool.
 */
for (i = 0; i < TASK_POOL_COUNT; i += 1) {
    worker_destroy(worker[i]);
    taskp_destroy(task_pool[i]);
}
/*
 * Destroy all message queues.
 */
for (i = 0; i < QUEUE_COUNT; i += 1)
    msgq_destroy(queue[i]);
/*
 * Finish the application by returning the value given by the
 * controller() at exit.
 */
    return c;
}

5.6 Performance

To characterize the efficiency of our approach, we have implemented the audioproc application using the Flow language with Feldspar and generated C code for it. We followed the specifications of the C run-time system described in Section to implement that over the POSIX threading library, pthreads. The sources were measured by sloccount\(^8\) (version 2.26, default settings), where the difference shown in the cost of implementation is out of question: our Flow-based version is below 200 lines – SLOC: 147, while the generated C code is above 1000 lines – SLOC: 1017. The size of the accompanying user code written in C is around of SLOC 89, while the running system is of SLOC 367.

The performance evaluation was both done on a Dell 2950 server (2x4 core Intel Xeon L5320 at 1.86 GHz) running FreeBSD/amd64 9-STABLE (of February 26, 2012), using the default FreeBSD system C compiler (GCC 4.2.1) and a TILEExpress-20G board (64-core Tilera TILE64) running Linux/tile64 2.6.26.7-MDE-2.1.2.112814, using a Tilera-enabled version of GCC. In both cases we compiled the C code with global compiler optimizations (-O2) enabled. Fortunately, both of them implement pthreads. However some of the operating-system-specific extensions had to be used to set processor affinity for the threads.

\(^8\)Available at http://www.dwheeler.com/sloccount/.
(a) A single pool with a single worker.

(b) A single pool with multiple (concurrent) workers.

(c) Multiple (declaratively partitioned) pools with multiple workers.

Figure 5.11: Performance results for running audioproc on a Dell 2950 with 2 hardware threads.
We measured the per-second throughput in kilobytes of the constructed dataflow network to see how efficiently scheduled it is. Note that the goal of the application is...
to maintain a speed of about 172 KB/s as that is the required bandwidth of a 44.1 kHz 16-bit stereo signal. Hence we modified the original application by counting the received bytes to learn how much data can flow through the network if there is no such limitations at the end. In other words, both the source and sink nodes have unlimited bandwidth. The data is transferred in 64-element chunks between nodes.

In addition, we did not use more than 2 hardware threads because the application can be conviently cut in to only two: both Figure 5.11 and Figure 5.12 indicate that it is still enough to achieve a stable double throughput. Having hardware threads is also important as our future goal is to be to deploy such graphs without the intervention of the pthreads library. The weak output experienced on the Tilera board is a consequence of that architecture featuring many parallel but individually weak processing units, i.e. it demands maximal parallelization of everything, and Feldspar currently does not support code generation for those boards (e.g. parallelization of loops).

In the measurements, we compared three different variations of the audioproc example (see Section 5.3). The first version (Figure 5.11a), Figure 5.12a) has a single pool with a single worker, i.e. that is a single-threaded application with round-robin scheduling between the tasks. It has a stable but low throughput.

In the second variation (Figure 5.11b, Figure 5.12b), we did not change the high-level program but naively launched 2 worker threads (in the automatically generated code) to execute tasks from the same (single) task pool. Although it starts up nicely, after a few seconds the performance dramatically collapses, even below the single-threaded case.

Thus an enhanced version is suggested to maintain both the high throughput and its stability. The results are displayed on Figure 5.11c and Figure 5.12c. Note that while the previous requires some implementation of mutual exclusion for the task pools, that latter does not, as each worker has its own pool to work with.

```haskell
pool0 = scheduleBy maxQlen (usePool 0)
pool1 = scheduleBy maxQlen (createPool 1)

audioproc' = split' --< (processLeft, processRight) --> merge'
  where
    split' = pool1 #= split
    merge' = pool0 #= merge
    processLeft = pool0 #= fft --> effectLeft --> ifft
    processRight = pool1 #= fft --> effectRight --> ifft
```

### 5.7 Summary

We provided a characterization of a type of systems employed in the field of digital signal processing in the chapter. During the investigation into how such a system might look in practice, we noted the following features: the task processes data from multiples sources with similar lines of processing, called pipelines. Pipelines are made from segments that may contain many different kernels that may be replaced dynamically at run time. Events are also supported because they are used in such systems to instruct the processing pipelines to change. As our summarized experience on the subject presented in Figure 5.1 shows: the input is first split into pieces to be routed to one of the pipelines then assembled after other end of the processing has finished.
To model such a system, we created a toy example of a simple audio processing application, called audioprocc. audioprocc is programmed as a combination of the combinators from the Flow language and Feldspar, where the latter is used as a domain-specific language to express the small programs inside the nodes or in the Flow terminology, kernels. The open nodes in this program were programmed as simple C programs that used the functions of the PulseAudio API. We could have given an example where no such user-level libraries were used, but we opted for them to illustrate the purpose of the functions to be provided by the user. Independently of this, we believe that similar functions can be written when lower level communication with the hardware devices is desired. As the associated performance measurements supported, the implementation can be indeed done in an efficient way, doubling the throughput of the constructed dataflow network on two processors.

Note that the Flow language here aimed to implement only the parts missing from Feldspar, everything else can be expressed on the level of the domain-specific language. That makes the Flow flexible as it simply assists to the partner language in the process of extension. Problems like serialization/deserialization (any other related protocol) of data communicated between nodes is left to the implementors of the domain-specific languages as part of the development of the binding. That makes it still lightweight enough to be used without interfering with the other languages.

On the other hand, delegating that degree of freedom to the domain-specific language has shown the advantage of extending the deforestation properties of Feldspar to the larger program. Depending on how the kernels are marked within the program, it becomes possible to remove the intermediate data structures (e.g., messages queues) between nodes. That does not require the Flow to know anything about the specifics of the language used. Thus, if the user connects two Feldspar programs sequentially then they may be fused together, while if the same is implemented in the Flow language, the channel is inserted and their execution may get overlapped and not fused. Since the complete program is written up by only a few expression, it is easy to reorganize the parts. Everything else may be then generated (derived) automatically.

As another backend for the Flow program, we have shown how to simulate the behavior through the semantics of the abstract program. Because each input data (i.e., the information coming through the source node, and the global parameter set for the kernels) was represented as an infinite list, they can be viewed as a series of values. The Cartesian product of the sets characterized by the infinite lists results in a series of discrete states that enables a discrete simulation for the constructed model.
Chapter 6

Related Work

Madhavapeddy et al. [47] targets a similar goal to ours, their work, Mirage is designed and implemented with functional programming in mind. The problem of how to deal with the platform-dependent details of running a system written in a functional language on top of hardware is elegantly delegated to hypervisors and only a port of the OCaml run-time system has to be provided for them. In contrast, Haskell is only used as a flexible and powerful specification and meta-language in our solution. The result would always be some target-language, e.g. C or even LLVM code. We are confident that it has the advantage over the approach of Mirage that no run-time system has to be ported. It is sufficient only to provide the set of abstractions for the given platform that we are depending on, that may already be present already there. However, porting a run-time system of an existing and well-established programming language gives more freedom to the programmer as he is technically able to use any pre-existing construct in the language. The question of which method will provide more benefits in the long run cannot yet be judged in our opinion. Apart from that, we agree with the concept of Mirage, that is typically a domain-specific system where a functional language can compete with the standard (and sometimes bloated) approach. But we note that OCaml is not pure a language that may complicate the question of correctness for Mirage.

A similar attempt for Haskell can be also experienced from the side of HaLVM [22] where they essentially chose the same approach and ported a Haskell run-time system to the top of a Xen abstraction layer. Running and working with Haskell in the field of systems programming is otherwise a constant topic of the joint project of Galois, Inc. and the Portland State University, titled “High-Assurance Systems Programming” (HASP) [31]. They have three major research goals: they want to implement a brand-new functional programming language, called Habit [30], that is directly engineered towards systems programming; a high-assurance run-time system [50]; development of tools and techniques for formal reasoning about each of these and their interconnection. Based on the available results of the project their work is heavy-weight compared to our approach as they are trying to solve the aforementioned problems in general with mathematical precision. The PhD thesis of Leslie [42] illustrates how they answer the question of implementing a memory-safe operating system in a purely functional language on the top of the L4 microkernel [45, 46]. In comparison, we have only barely touched the question of memory management as we do not assume dynamic but static memory management in the context of a well-defined (and restricted) computation model. It is a trade-off in favor of making our model simple and manageable. The audiostream example also shows that our advantage is in gearing the constructed model towards a certain application
domain, and we are not trying to solve all the problems in general. The home page [31] currently shows a minor decline in the progress that lead us to believe that it is hard topic. Perhaps our approach may be tackled better for those specific cases.

In the past, the researchers from the same group worked with the House operating system [28], which is implemented in Haskell. In Granuke’s master’s thesis [26] a branch of House is introduced, titled Lighthouse which integrates the Lightweight Concurrency framework [43] for experimentation. As a result, the thesis features an extensible operating-system scheduler programmed in Haskell. It is similar to our thoughts as it tries to raise the process of development of schedulers to an abstract level. However, it still closely adheres to the traditional foundations. The implemented scheduler is basically passive, that is, it may be considered rather as a collection of routines written in Haskell. There is a simple interface defined but not for hints as we presented in our thesis. One of the main reasons is that we have chosen a cooperative concurrent multitasking without supporting preemption. Nonetheless it is interesting to learn that House originally lacked the notion of thread priority. To address the problem, Lighthouse utilizes that feature of the incorporated concurrency framework and extends the handling of priorities in many different ways. That also spawns the extensible scheduler framework referred to above which facilitates straightforward implementation of various scheduling policies. In our case, no concept of priority is introduced as competing program codes are wrapped into tasks and the user himself is offered the opportunity to organize the optimal execution scheme for the Flow program.

The recent works of Simon Marlow et al. [48, 49] show that expressing workflow systems in functional languages is still very much a hot research topic indeed. The Par monad is an extensive and generic tool to support parallel programming in a very efficient way. It does not do any I/O hence it is considered pure and therefore it can be used at many different places to describe similar (even dynamic) dataflow networks, where a scheduler can be also specified, but there the scheduler interface of the Par monad is not for general consumption, rather only provided as an “escape hatch” for relaxing certain cases. However, it uses many tricks (like IORefs) to make it work inside Haskell and is not concerned with code generation. Flow, on the other hand, tries to avoid most monadic features and concentrates instead on how to build automatically generated programs supported by a minimalist run-time system. It then can be used for describing and compiling event-driven system programs for embedded hardware.

Hernýák et al. [32] work with a coordination language, called D-Box (together with D-Clean), for Clean programs which is similar to our proposed framework (if we abstract away from the differences between Haskell and Clean). However, their focus is on supporting distributed applications. We believe that similar skeletons may be easily constructed in the Flow language, simply written as higher-order functions (or meta-language templates). As we have decided to employ language embedding as an implementation technique and they implemented a standalone compiler for translating their high-level programs, they had to cope with the connected typing and implementation problems. Their work also puts emphasis on supporting and enforcing channel protocols while in our work these are simply delegated to the little languages used in the network. As an additional note, the Flow language turns out to be general as depending on the attached run-time system, it is easy to provide support for implementing network communication between the nodes, therefore it may be used for describing distributed computations. Although we have to pay the price that source and sink nodes cannot be described by the Flow programs. Whereas in D-Box an explicit support for that is implemented. Another
important difference is that D-Box has better support for dynamic behavior in the form of dynamic channels.

It is typical for embedded systems that the operating system is prepared to be deployed on the given hardware. Solutions based on microkernels provide some support for this. A prominent representative of this approach is Enea OSE [15] which is one of the most widely used real-time operating system in the industry. The primitives featured in Enea OSE architecture are very similar to the one we capture in our model (processes with message passing), and the implementation is highly sophisticated: modular, layered, fault-tolerant, distributed, event-driven, deterministic architecture with task monitoring and optimized memory usage. A promising attempt is ArchiDeS (Architecture, Deployment, Scheduling) [6], which is a research framework written in C++ for building a large stream-processing system on multi-core processors. It supports run-time configuration of the constructed application. The key concepts for ArchiDeS are the interface ports and interface port types, containing a dedicated message handler to specify the run-time behavior for the given port. Interfaces can be assigned to single or shared component modules that are first-class entities in the system. There is also a replaceable scheduler and a run-time system paired up with the components that supports different, large-scale multi-core chips and application-specific scheduling. It features both data and pipeline parallelism, similar to the solution presented here. However, a drawback of those tools from our approach is that they still have to be programmed in C or C++. The modular design provides nice abstractions as building blocks, but due to the nature of such programming languages it is hard for the compiler to figure out how to optimize the constructed applications further, like removing intermediate data structures when they are not needed for performing similar simplifications in the application. Besides that, the application code still has to be written in C or C++ which tends to be more error-prone and verbose compared to high-level and domain-specific languages.

The use of domain-specific languages for generating code for operating systems is a well-known technique. The Barrelfish operating system is the result of a research project into the possibilities in structuring operating systems for today’s and future’s hardware. Barrelfish features a development framework, named Filet-of-Fish where the researchers have chosen a similar approach to ours: essentially, they embedded C into the functional language Haskell. Filet-of-Fish gives strong static guarantees that the code generated is valid by construction and it can be always compiled. In contrast to our approach, both Barrelfish and Filet-of-Fish solve the problem for generic operating systems, while we are focusing only on concepts that are specific to a domain.

From the side of digital signal processing, it is still an open research problem to find an elegant way to describe a digital signal processing system in a common high-level language to achieve productivity, efficiency, and portability at the same time [55]. A distinct characteristic of digital signal processing algorithms is that they usually do not require run-time decisions, they can be expressed as so-called signal-flow graphs. In signal-flow graphs, computations are represented by graph nodes and dependencies between the computations by branches, and they can be used for analysis and code generation, making this formulation popular for modeling such systems. That also supports our way of representation since we are technically building such graphs in the Flow language.

As a related project, Ptolemy [41] studies modeling, simulation, and design of concurrent, real-time, embedded systems, focusing on assembly of concurrent components, and using well-defined models of computation that govern the interaction between the components. Ptolemy is based on the principles of object-oriented programming and it has a
decent implementation in Java. Although it solves many problems (e.g. scheduling) related to the development of operating systems for digital signal processing applications, it still can be only considered a generic research effort into finding an appropriate modeling language for such systems, and not a way in which to provide reliable and clever compilation of components, which is our focus. Nevertheless, results of the Ptolemy project can be re-used here to handle some of the computation-related aspects of the elements to be modeled resolved.
Chapter 7
Conclusion and Future Work

This document summarizes our attempt to use a contemporary functional language, Haskell, as a specification language and develop a compiler for it. We used the technique of language embedding to create a glue language, called Flow, to implement a method for composing programs written in arbitrary embedded domain-specific languages. During the discussion, we defined certain conventions and conditions that assist in construction of applications to be run directly as part of embedded systems. We used Feldspar, an embedded domain-specific language as an example to demonstrate how to construct an application in the domain of digital signal processing. Digital signal processing is an important domain in the industry and with the advent of many- and multi-core processors, it constantly searches for new technologies for developing systems in a rapid, yet reliable and suspendable way.

The Flow language was designed with requirements of such systems in mind. Certainly, we believe that the Flow language cannot be used just for building digital signal processing systems, but it offers enough flexibility to be used in other domains, or even with algorithms of multiple domains at the same time. Regarding this, Flow gives freedom to the authors of those little languages as it has only minimal requirements on the implementation, and it makes possible to extend it further inside the given language to reach our goals (as we presented in Chapter 5).

Based on the experiences during our research in the area, we can draw the following conclusions.

– Domain-specific nature of the description provides better visibility to the compiler that can be exploited during optimization and code generation. If the constructs for a program are too low-level, then the compiler must be very intelligent to identify abstractions in the source program. Another danger of being low-level is that it is easy to write programs that are hard to reason about, hard to determine whether they conform to the specification. Domain-specific languages, on the other hand, usually offer more high-level constructs that help to guide the thoughts of the programmer around a given abstract model that captures the main concepts of the domain. By using the domain-specific constructs, the programmer is able to reveal more of his intentions, and thus the compiler is able to learn more about the problem to be solved. In a fortunate case, the definition of the language itself coincides with a formal specification, i.e. that programs themselves become specifications. In case of Feldspar, they are the mathematical formulas that the corresponding programs shall implement as illustrated in Chapter 2, and supported by [8, 9].
Declarative nature of the description helps the programmer to avoid the error-prone and uncreative work of writing boilerplate code. Restructuring the application requires less effort. From the point of view of compilation, the declarative approach technically contributes to the specification being viewed as input to a knowledge base, wherein the compiler has some freedom in the translation as it knows more about the system, and it knows this on a higher level. The declarative view also motivates the programmer to not to be too loose in the details of the solution. Such way of extracting the essentials makes the resulting programs compact – similar to the expression representing the audioproc application in Chapter 5 –, and therefore easy to understand and change. That is described in [54].

Compositionality pays off in development of operating systems as well. Choosing to express an operating-system-like program as a composition of multiple layers implies a powerful software engineering methodology. Mixed with the declarative approach, composition – or merging – of the layers may profit from the removal of intermediate data structures, and that way they may be fused to a specialized program whilst it was being written as a composition of generic components. We believe that the Flow language presented in Chapter 3 and [53] is a good example of how to design and implement an abstract vehicle that may be used as an additional layer to re-use existing components in a different setting.

The domain-specific semantics demands only a specialized minimal run-time environment that can be ported to several architectures without major difficulties. In Chapter 3 we have presented the semantics of the foundations for representing and running programs as dataflow networks that operate only with a few abstractions: task pools, tasks, message queues, and workers. As part of Chapter 5, we have given a concrete example on the API required to be implemented for the C programming language on top of a POSIX-compliant system. We are confident that it may be lowered down to the bare metal, though there are embedded system manufacturers (e.g. Tilera) that already provide a POSIX-compatible run-time environment on top of their boards. The results were published in [54].

In summary, the observations above enable us to conclude that the potential of functional languages in compiler technology come with certain advantages in the development of operating systems, where models may be taken as specifications for reliable code generation.

7.1 Future Work

However, there are some evident points in our current research that may be concluded even further as part of a potential future work.

In our opinion, it is possible to continue with finding and characterizing additional higher-order combinators for selector functions to cover a wider class of schedulers. In the thesis, only the concept and a few examples were presented, but we believe that area still has some research potential. We left it for future work as our goal was rather to show a proof of concept.

Nor we did discuss the algorithm for finding automatic partitioning and selecting the optimal selector function for task pools. In the current model, the user (i.e. the programmer) is responsible for manually drawing the lines between the pools and assigning
selectors to them. It may be misleading, but we have noted that if there is no annotation present in the high-level description about these settings, only a default scheme will be applied, which actually means employing some trivial heuristics. In our opinion, our current results presented in this thesis may serve as a good starting point to experiment with and implement such an algorithm as it technically establishes some basic combinators that the user can use, but the program still cannot. It is a deep and interesting topic that has also been left for further research.

It would be worthwhile exploring how to extend the Flow language with support for more dynamic behavior, e.g. introduce the optionality property of channels. However, at the moment it may be expressed with a Maybe-like FlowType type, where a data equal to Nothing is sent over the channel to signal that in the current computation step there is no data coming from that direction so the program within the connected node may skip its processing. But that may not be optimal in terms of performance, therefore it is desirable to investigate how to get rid of the imposed overhead in the generated code.
Acknowledgments

We would like to acknowledge the financial support of Ericsson Software Research, Ericsson Business Network Units and SSF (Sweden), the Hungarian National Development Agency (KMOP-2008-1.1.2), Programul Operațional Sectional Dezvoltarea Resurselor Umane 2007–2013 (POSDRU/6/1.5/S/3-2008, Romania). We would like to thank the staff members and students of the Feldspar Group, Department of Programming Languages and Compiler at Eötvös Loránd University, Department of Computer Science and Engineering at Chalmers University of Technology, and Department of Computer Science at Babeș-Bolyai University who contributed to the development of this thesis and the related research work with their comments and work. Furthermore, we would like express our gratitude for being actively involved in the The FreeBSD Project, where we are responsible for maintaining the Glasgow Haskell Compiler and the associated Haskell Cabal packages, equally giving us insights in contemporary operating systems and functional programming at the same time. It is similarly important for us that we had the opportunity to teach Haskell on the Eötvös Loránd University at beginner and advanced levels for many semesters. Finally, we are especially thankful for K.M.M. for the extensive proofreading.
References


