BABEŞ-BOLYAI UNIVERSITY, CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

EÖTVÖS LORÁNT UNIVERSITY, BUDAPEST
FACULTY OF INFORMATICS

Functional Modelling of Operating Systems

Abstract of Doctoral Thesis

PÁLI Gábor János
Doctoral Student

Horia F. POP (BBU)
KOZSIK Tamás (ELTE)
Supervisors

Cluj-Napoca, March 2012

Contents

Contents of the Thesis

Acknowledgments

References

Keywords

Keywords: Functional Programming, Embedded Systems, Operating Systems, Dataflow Networks, Scheduling, Declarative Programming, Language Embedding, Haskell, Domain-Specific Language, Systems Programming, Reactive Programming, Executable Semantics, Pure Semantics, Code Generation, Compiler, Thread Pooling, Task Parallelism, Tacit Programming

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". 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 of software themselves.

Nowadays a serious obstacle in the spread of an operating system is lack of support for it. Without proper hardware support 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. Quick evolution of hardware also forces the programmers to think on a higher level, where the associated compiler becomes their partner. Such a typical field of application of that methodology is tied to various domains, where more semantical information can be extracted from the specification, therefore better target-language programs can be generated with less human intervention.

This 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 domain-specific programming languages. The purpose of the `Flow` is to turn Haskell into a specification language that is compact yet expressive enough to describe complex applications as a composition of domain-specific-language programs.

– 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 are 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.

In Chapter 1, we provide an overview of the fundamental problems in this area of research, a brief description of the probleam 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 being 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 of that, *Feldspar*, a domain-specific language for digital signal processing and parallelism is discussed briefly to illustrate the concepts of embedding.

In Chapter 3, a glue language for little languages called `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 `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 `Flow` language. The problems discussed include the way in which the components are scheduled for execution by distributing the available resources 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. In addition to this, a user-controllable way of scheduling is defined that helps to parallelize the aforementioned abstract machine.

In Chapter 5, the discussion continues on `Flow` by taking an advanced example application and demonstrate the design. However, our further intention with this application is to explicitly characterize and thus model a given 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 – and it is limited to a specific application domain, 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.

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

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 discove 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. This is still an open question since high-levle, especially functional programming languages are not yet wide-spread and are not currently adapted to everyday industrial use.

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 are usually stronger and more strict than the one created for C++ therefore they simply do not allow the programmer to mix and match different weights of abstractions at her easer. 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 to be preferred as vehicles for operating systems development. Many popular modern and high-performance systems, like the Linux-based ones, or FreeBSD are still developed in C.

Apart from performance, the security and reliability of operating systems are frequently considered. It is easy to write high-performance but unsecure, 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 most 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 most of the error-prone details of the source code is derived, generated, or managed automatically, and because the syntactical constructs are usually closer to the exact program specification.

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, i.e. 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.

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 high-level programming languages already and their source code is mostly open for study. That is, 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.

2 Employed Techniques

In this chapter we present the techniques which we employed to achieve our goal: as we 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, restriceted languages on top of its basic constructs. Haskell itself is built up in a similar fashion – that is not accidental too: in the chapter, there are examples given of why such a structure is favorable in the definition of languages or even any complex application programming interface.

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 even specification, language inherits many of the properties of the embedding, i.e. the host language. In case of Haskell, this is the function-oriented nature or purity, for example. 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 versus backend" separation wher 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.

To demonstrate the concepts and introduce a contemporary embedded language in Haskell for later use, we are going to mention *Feldspar* [2, 8, 9] as an example here. *Feldspar* is a high-level language that allows description of signal processing algorithms, and it is the result of a joint research project of Ericsson, Chalmers University of Technology (Göteborg, Sweden), Eötvös Loránd University (Budapest, Hungary).

```
sumSquares :: Data Int -> Data Int
sumSquares n = (sum . map square) (1 ... n)
  where square x = x * x
```

Figure 1: Feldspar program `sumSquares`

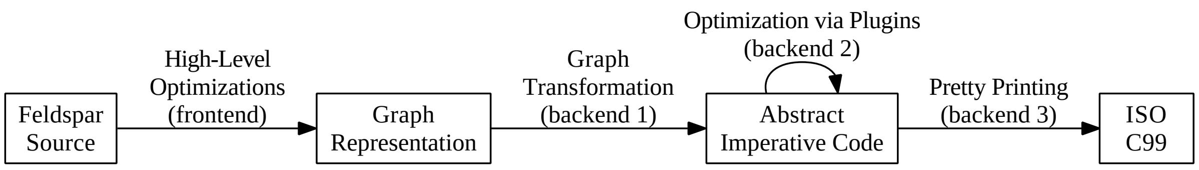


Figure 2: Internal structure of the *Feldspar* compiler.

```
#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 3: Final, optimized C version of `sumSquares`.

These programming techniques are introduced to prepare the presentation of the main contributions of the thesis where we use Haskell as a specification language. Note that 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 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 back-ends. 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 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.

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 this chapter, there are three different layers of languages used: 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 in the chapter, dataflow networks are first represented by an algebraic data type with little programs in its nodes which then mapped to a mathematical graph.

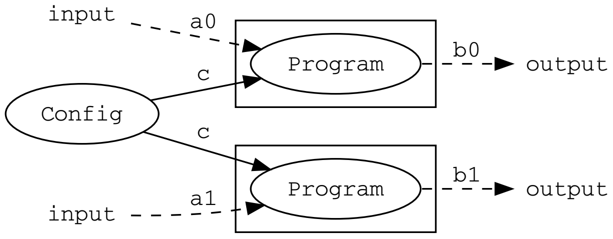
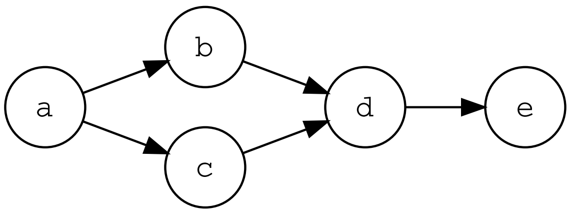


Figure 4: Nodes in a `Flow` network: domain-specific programs with associated global parameters.

The following graph:



can be described by this program:

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

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.

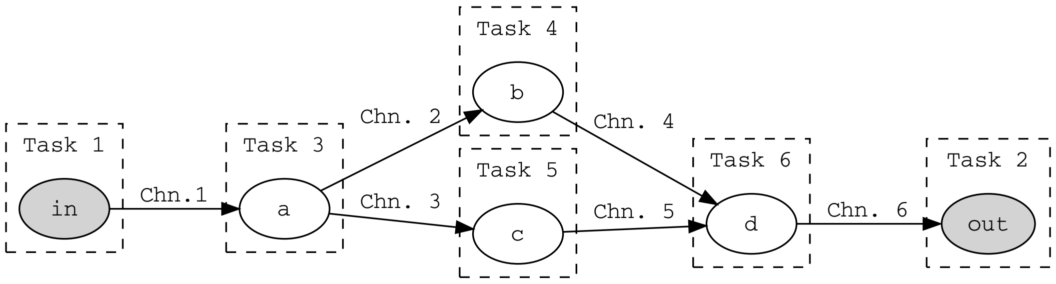


Figure 5: Decomposing a `Flow` into tasks and channels, called an *abstract program.*

In parallel with that, we 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.

The goal of the framework 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. Thus we call those graphs *flows*.

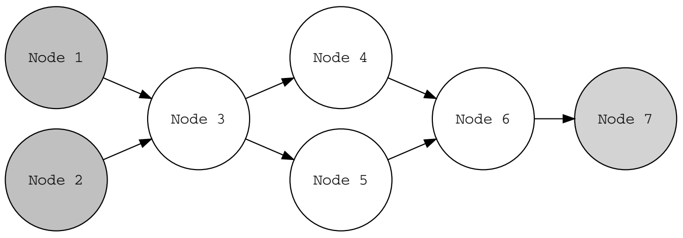


Figure 6: Example of a flow. Dark grey nodes (1 and 2) are source nodes, and the grey node (7) is a sink node.

An example for a flow is shown in Figure 6. 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 node*s are the nodes that do not have any preceding node (parent) in the graph. *Sink node*s are the nodes that do not have any suceeding 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 her 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.

We 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.

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.

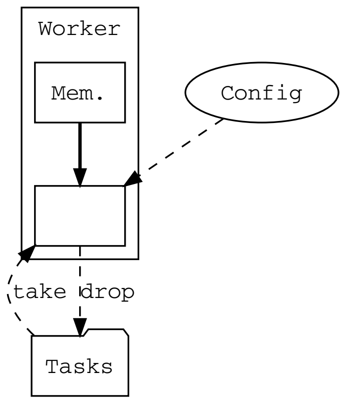


Figure 7: Graphical view of a worker with tasks replaceable in run time.

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 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.

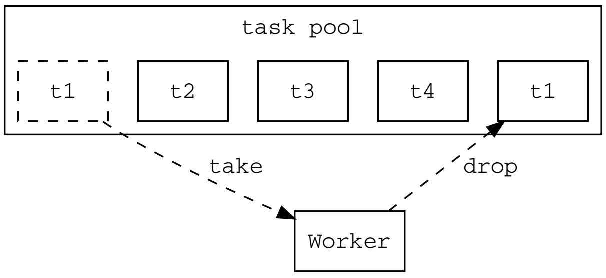


Figure 8: Operation of a single worker over a task pool.

It is then 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 is also be taken under consideration.

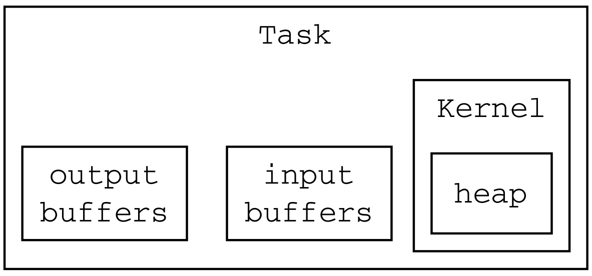


Figure 9: Task memory layout in general.

As we 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 help us to define a parallel version of the previously introduced abstract programs and the associated machines. We thereby extend the semantics and the simulation of `Flow` programs for parallel execution.

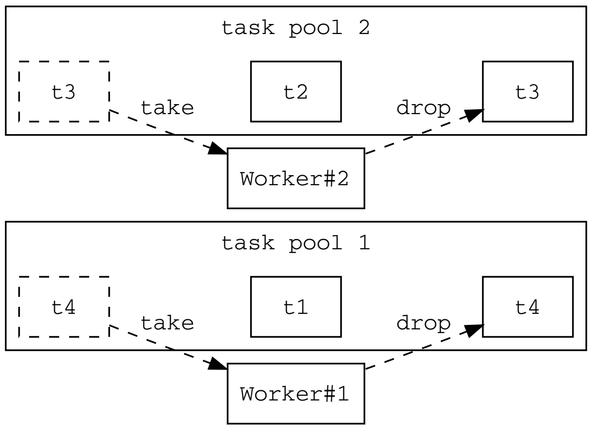


Figure 10: Operation of multiple workers over multiple task pools.

For the remainder of the chapter, the code generation for `Flow` programs is discussed. By explaining the details of the proposed code generation procedure, the previously omitted components of the `PrimKern` type class is 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` is 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 turns 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 is explained. In connection with the nodes without a program we also touch on the question of how to deploy a `Flow` system.

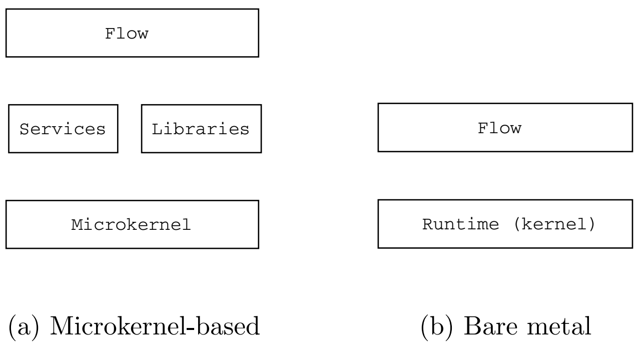


Figure 11: Scenarios for deploying a `Flow` system.

We present 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 stucks 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.

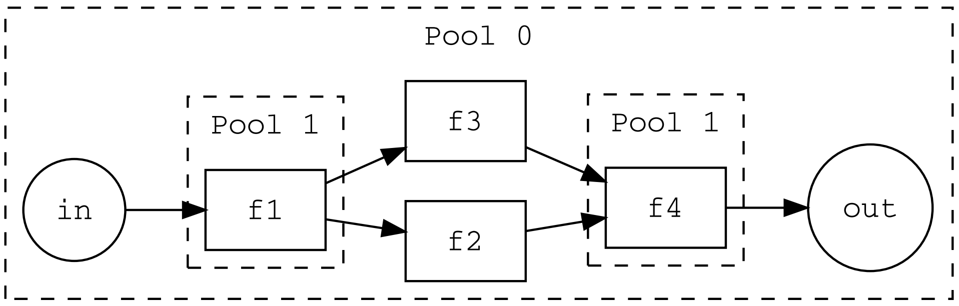


Figure 12: Programmer-defined partitioning of tasks into pools.

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. This 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 are 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. Simplicity 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.

5 `Flow` Programs as Operating Systems

In order to validate the `Flow` terminology and methodology introduced in the previous chapters, we 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*. 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.

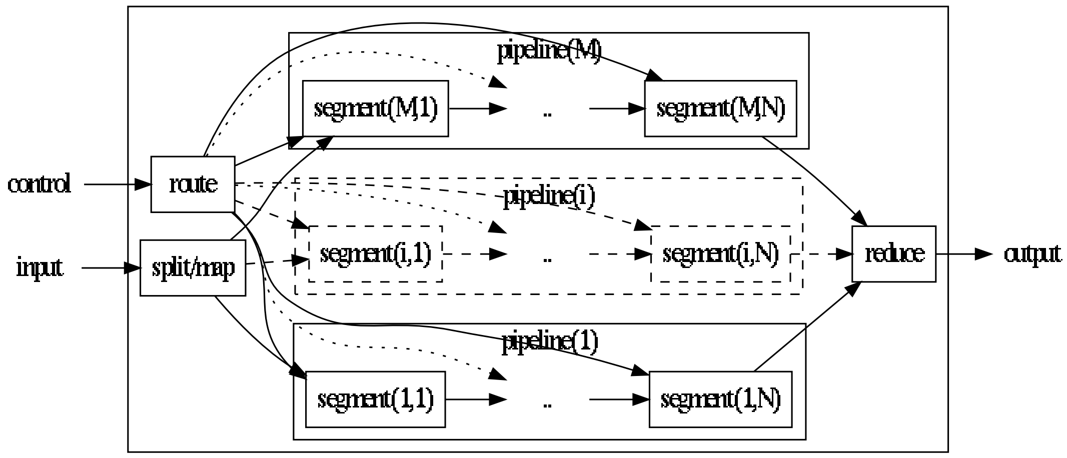
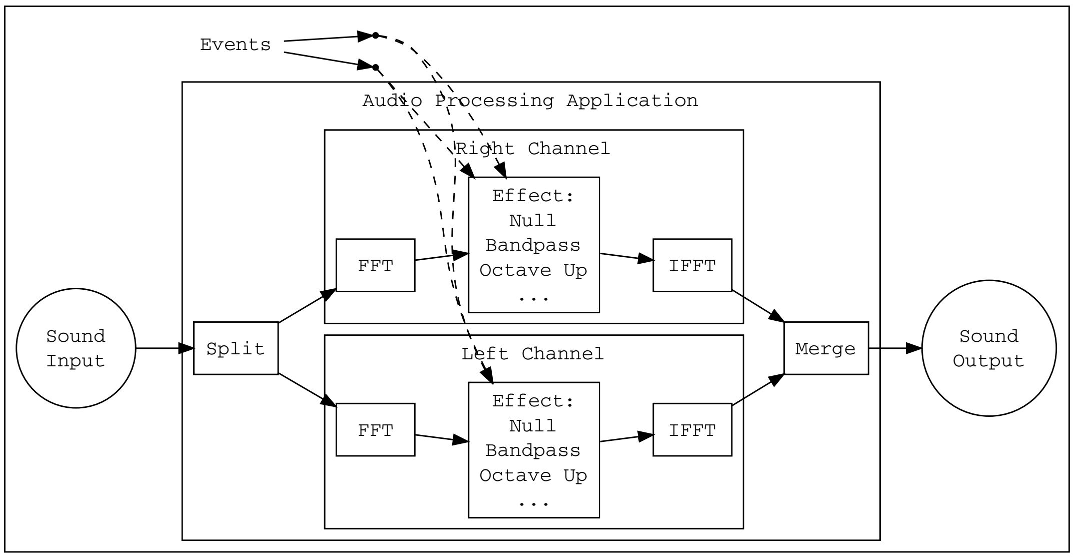


Figure 13: Abstract model of a digital signal processing system.

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. Barrelfish [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.



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

Figure 14: A sample audio processing application in `Flow` and *Feldspar*.

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 simulatenously 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.

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 for 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 take the opportunity to walk through those details as well.

Note that, for the sake of clarity and simplicity, we do 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 are 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.

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 13 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 `audioproc`. `audioproc` 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 [24], 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.

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 `audioproc` 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.

Hernyák 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 in to 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 code 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.

7 Conclusion and Future Work

This thesis 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 embeddeding 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.

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 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 –, 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 [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 the thesis, we present 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 the thesis, 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.

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 thas 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.

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 Operaţional 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 Babeş-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

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, Y. Michael. *Mach: A New Kernel Foundation for UNIX Development*, Proc. of the USENIX 1986 Summer Conference, June 1986.

[2] E. Axelsson, G. Dévai, Z. Horváth et al. *Feldspar: A Domain-Specific Language for Digital Signal Processing Algorithms*, Proc. 8th ACM/IEEE International Conf. on Formal Methods and Models for Codesign, MemoCode, IEEE Computer Society, 2010.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. *Xen and the Art of Virtualization*, Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP), New York, New York, 2003.

[4] M. Ben-Ari. *Principles of the Spin Model Checker* (1st Ed.), Springer, 2008.

[5] W.R. Bevier, L. M. Smith. *A Mathematical Model of the Mach Kernel*, Technical Report 102, Computational Logic, Inc., 1994

[6] M. Brorsson, K-F. Faxen, K. Popov. *ArchiDeS: A Programming Framework for Multicore Chips*, Swedish Workshop on Multicore Computing (MCC09), 2009.

[7] P.E. Dagand, A. Baumann, T. Roscoe. *Filet-o-Fish: Practical and Dependable Domain-Specific Language for OS Development*, ACM SIGOPS Operating Systems Review, 2010.

[8] G. Dévai, Z. Gera, Z. Horváth, G. Páli et al. *Feldspar – A Functional Embedded Language for DSP*, 8th International Conference on Applied Informatics (ICAI2010), 2010.

[9] G. Dévai, M. Tejfel, Z. Gera, G. Páli et al. *Efficient Code Generation from the High-Level Domain-Specific Language Feldspar for DSPs*, Proc. ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems, assoc. with IEEE/ACM International Symposium on Code Generation on Optimization (CGO), 2010.

[10] I.S. Diatchki, M.P. Jones, R. Leslie. *High-Level Views on Low-Level Representations*, Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, 2005

[11] I.S. Diatchki, M.P. Jones. *Strongly Typed Memory Areas Programming Systems-Level Data Structures in a Functional Language*, Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell, 2006

[12] I.S. Diatchki. *High-Level Abstractions for Low-Level Programming*, PhD thesis, OGI School & Engineering, Oregon Health & Science University, 2007

[13] I.S. Diatchki, T. Hallgren, M.P. Jones, R. Leslie, A. Tolmach. *Writing Systems Software in a Functional Language: An Experience Report*, Proceedings of the 4th Workshop on Programming Languages and Operating Systems, 2007

[14] C. Elliott, S. Finne, O. de Moor. *Compiling Embedded Languages*, Proc. Semantics, Applications and Implementation of Program Generation (SAIG 2000), 2000.

[15] Enea AB. *Enea OSE: Multicore Real-Time Operating System*, `http://www.enea.com/`, 2011.

[16] Feldspar Group. *Feldspar Home Page*, `http://feldspar.inf.elte.hu/` Date: 05-26-2011.

[17] The `feldspar-language` package. `http://hackage.haskell.org/package/feldspar-language-0.4.0.2`, Date: 04-27-2011.

[18] The `feldspar-compiler` package. `http://hackage.haskell.org/package/feldspar-compiler-0.4.0.2`, Date: 04-27-2010.

[19] M. Felleisen, B. Findler, M. Flatt et al. *Building Little Languages With Macros*, Dr. Dobb's Journal, April 2004.

[20] G. Fu. *Design and Implementation of an Operating System in Standard ML*, Master's thesis, University of Hawaii at Marona, 1999

[21] W. Fu, C. Hauser. *A Real-Time Garbage Collection Framework for Embedded Systems*, ACM SCOPES, 2005

[22] Galois, Inc. *HaLVM Home Page*, `http://www.halvm.org/`, Date: 01-15-2012.

[23] Z. Gilián. (supervisors: Z. Horváth, G. Páli) *Abstract Description of System-Level Layers in a Functional Language*, BSc. thesis, Eötvös Loránd University, Faculty of Informatics, 2010.

[24] A. Gill, J. Launchbury, S.L. Peyton Jones. *A Short-Cut to Deforestation*, Proc. Int. Conf. on Functional Programming Languages and Compiler Architecture (FPCA), 1993.

[25] M. Golm, M. Felser et al. *The JX Operating System*, Proceedings of the 2002 USENIX Annual Technical Conference, Monterey, CA., 2002

[26] K. Granuke. *Extensible Scheduling in a Haskell-Based Operating System*, Master's thesis, Portland State Universtity, 2010.

[27] T. Hallgren. *Fun with Functional Dependencies*, Proc. of the Joint CS/CE Winter Meeting, Varberg, Sweden, January 2001.

[28] T. Hallgren, M.P. Jones et al. *A Principled Approach to Operating System Construction in Haskell*, The 10th ACM SIGPLAN International Conference on Functional Programming, 2005

[29] T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy. *Composable Memory Transactions*, Proc. of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), Chicago, Illinois, June 15–17, 2005.

[30] The High-Assurance Systems Programming Project (Hasp). *The Habit Programming Language: The Revised Preliminary Report*, November 2010.

[31] The High-Assurance Systems Programming Project (Hasp). *Hasp Home Page*, `http://hasp.cs.pdx.edu/`, Date: 01-31-2012.

[32] Z. Horváth, Z. Hernyák, V. Zsók. *Implementing Distributed Skeletons Using D-Clean and D-Box*, Proc. of the 17th International Workshop on Implementation and Application of Functional Languages (IFL), Dublin, Ireland, September 19–21, 2005, pp. 1–16.

[33] P. Hudak, J. Hughes, S.L. Peyton Jones, P. Wadler. *A History of Haskell: Being Lazy with Class*, Proc. of the 3rd ACM SIGPLAN Conf. on History of Programming Languages (HOPL III), 2007.

[34] J. Hughes. *Why Functional Programming Matters*, The Computer Journal – Special Issue on Lazy Functional Programming, Vol. 32, Issue 2, Oxford University Press, April 1989.

[35] G.C. Hunt, J.R. Larus. *Singularity: Rethinking the Software Stack*, ACM SIGOPS Operating Systems Review, Vol. 42., No. 2., pp. 37–49., 2007

[36] D.J. King, J. Launchbury. *Lazy Depth-First Search and Linear Graph Algorithms in Haskell*, Glasgow Workshop on Functional Programming, 1994.

[37] G. Klein, R. Kolanski. *Formalising the L4 Microkernel API*, Proceedings of the 12th Computing: The Australasian Theory Symposium, 2006

[38] G. Klein, P. Derrin, K. Elphinstone. *Experience Report: seL4 – Formally Verifying a High-Performance Microkernel*, Proceedings of the 14th International Conference on Functional Programming, 2009

[39] G. Klein, K. Elphinstone et al. *seL4: Formal Verification of an OS Kernel*, Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2009

[40] C. Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization*, MSc. thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[41] E. Lee, C. Hylands, J. Janneck et al. *Overview of the Ptolemy Project* Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001.

[42] R. Leslie. *A Functional Approach to Memory-Safe Operating Systems*, PhD thesis, Portland State University, 2011.

[43] P. Li, S. Marlow, S.P. Jones, A. Tolmach. *Lightweight Concurrency Primitives for GHC*, Haskell Workshop 2007

[44] D. Licata, S. Peyton-Jones. *View Patterns in GHC*, AngloHaskell 2007.

[45] J. Liedtke. *Improving IPC by Kernel Design.* Proc. of the 14th ACM Symposium on Operating System Principles (SOSP), pp. 175–188, 1993.

[46] J. Liedtke. *On Micro-Kernel Construction.* Proc of the 15th ACM Symposium on Operating System Principles (SOSP), pp. 237–250, 1995.

[47] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, J. Crowcroft. *Turning Down the LAMP: Software Specialization for the Cloud*, Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), Boston, Massachusetts, 2010.

[48] S. Marlow. *Parallel and Concurrent Programming in Haskell*, Central European Functional Programming Summer School (CEFP), Eötvös Loránd University, Budapest, June 14–24, 2011.

[49] S. Marlow, R. Newton, S.L. Peyton Jones. *A Monad for Deterministic Parallelism*, Haskell '11: Proc. of the 4th ACM SIGPLAN Symposium on Haskell, Tokyo, Japan, ACM, 2011.

[50] A. McCreight, T. Chevalier, A. Tolmach. *A Certified Framework for Compiling and Executing Garbage-Collected Languages*, Proc. of 2010 ACM International Conference on Functional Programming, Baltimore, September 2010.

[51] M.K. McKusick, G.V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*, Boston, Mass.: Addison-Wesley, 2004

[52] R. Milner, M. Tofte, R. Harper, D. MacQueen. *The Definition of Standard ML (Revised)*, MIT Press, 1997.

[53] G. Páli. *Extending Little Languages into Big Systems.* To appear in Horváth Z., Zsók V. (Eds.): Central European Functional Programming School. Fourth Summer School, CEFP 2011. Revised Selected Lectures. Lecture Notes in Computer Science, (ISSN 0302-9743), Vol. 7241, pp. 295–312.

[54] G. Páli. *Declarative Scheduling of Dataflow Networks*, Annales Universitatis Scientarium Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica, Vol. 36(2012), 2012 *(to appear).*

[55] G. Paller. *Rafael: An Intelligent, Multi-Target, Signal-Flow Compiler*, PhD thesis, Technical University of Budapest, Department of Electromagnetic Theory, June 1995.

[56] R. Paterson. *Arrows and Computation*, The Fun of Programming, pp. 201–222, Palgrave, 2003.

[57] S.L. Peyton Jones, J. Launchbury. *State in Haskell*, Lisp and Symbolic Computation 8(4), pp. 293–341, 1995.

[58] S.L. Peyton Jones, D. Vytiniotis, S. Weirich, G. Washburn. *Simple Unification-Based Type Inference for GADTs*, Proc. of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 50–61, Portland, Oregon, 2006.

[59] R. Pike, D. Presotto, K. Thompson, H. Trickey. *Plan 9 from Bell Labs*, Proc. of the Summer 1990 UKUUG Conference, 1990.

[60] R. Pike. *Systems Software Research is Irrevelant*, 2000.

[61] PulseAudio Home Page. `http://www.pulseaudio.org/`, Date: 10-20-2011.

[62] D. Rémy, J. Vouillon. *Objective ML: A Simple Object-Oriented Extension of ML*, ACM Symposium on Principles of Programming Languages (PLOP), 1997.

[63] D.M. Ritchie, R. Pike et al. *The Inferno Operating System*, Bell Labs Technical Journal, Vol. 2., No. 1., Winter 1997, pp. 5–18., 1997

[64] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, G. Heiser. *Automatic Device Driver Synthesis with Termite*, Proceedings of the 22nd ACM Symposium on Operating System Principles, 2009

[65] F. Schneider, G. Morrissett, R. Harper. *A Language-Based Approach to Security*, Informatics: 10 Years Back, 10 Years Ahead, 2000

[66] J. Shapiro, S. Weber. *Verifying Operating System Security.* Technical Report MS-CIS97–26, University of Pennsylvania, Philadelphia, PA, USA, 1997.

[67] A. Schüpbach, S. Peter, A. Baumann et al. *Embracing Diversity in the Barrelfish Manycore Operating System*, Proc. of the Workshop on Managed Many-Core Systems (MMCS08), June 2008.

[68] A.S. Tanenbaum, A.S. Woodhull. *Operating Systems: Design and Implementation, Third Edition*, Prentice Hall, 2006

[69] S. Thompson. *Type Theory and Functional Programming*, Addison-Wesley, 1991.

[70] L. Torvalds. *Linux: A Portable Operating System*, MSc. thesis, University of Helsinki, Department of Computer Science, 1997.

[71] B. Torma. (supervisor: G. Páli) *Development of a Framework for the Simulation of Real-Time Systems in a Functional Language* BSc. thesis, Eötvös Loránd University, Faculty of Informatics, 2011.

[72] A. Vajda. *Programming Many-Core Chips* (1st Ed.), Springer, 2011.

[73] A. Weelden, R. Plasmeijer. *Towards a Strongly Typed Functional Operating System*, Implementation of Functional Languages, 14th International Workshop, 2002

[74] D. Wentzlaff, A. Agarwal. *Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores*, ACM SIGOPS Operating System Review: Special Issue on the Interfaction among the OS, Compilers, and Multicore Processors, April 2009.