Functional Modelling of Operating Systems
Theses of the Doctoral Dissertation

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
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”. 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. 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 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 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 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.

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.

For example, Mirage [9] is the result of promising research that focuses on the domain of cloud computing, and features a specialized software stack for achieving that goal. Mirage is structured as a “vertical operating system” à 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 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 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.

Outline of the Thesis

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 different 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 called Feldspar (Functional Embedded Language for Digital Signal Processing and Parallelism) [7]. Feldspar features a compiler prototype which translates programs into hardware-independent ISO C99-compliant code, or recently, LLVM intermediate representation.

Software for DSPs is traditionally highly hardware-dependent and hence porting it to new processors usually requires significant design effort. 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.

In this thesis, Feldspar is used as prime example for a domain-specific language to be applied in conjunction with the Flow, while it is also exploited to introduce the basic concepts of contemporary language embedding in Haskell [8] that is also influenced the design and implementation of Flow.

A classic layout of complex software applications, such as operating systems, 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 grow, 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.

The Flow language is introduced 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.

During the work on the design and implementation of a coordination language framework for embedded domain-specific languages we briefly touch on the issue of executing specialized task graphs. In our model, we create 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 thesis 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
The thesis is structured as follows.

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 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.

In Chapter 3, Flow is described. 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 (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 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.

Conclusions

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.

**Thesis.** 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 and supported by [1, 2].

**Thesis.** 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, and therefore easy to understand and change. That is described in [6].

**Thesis.** 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 [4] 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.

**Thesis.** 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 give 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 [6].

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.

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.
References


